LATTICE GAS HYDRODYNAMICS WITH GALILEAN INVARIANCE

by

PETER A. DONIS

S.B., Nuclear Engineering
Massachusetts Institute of Technology
1987

SUBMITTED TO THE DEPARTMENT OF NUCLEAR ENGINEERING
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF SCIENCE IN NUCLEAR ENGINEERING

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
September 1988

Signature of Author .....
Department of Nuclear Engineering
September 16, 1988

Certified by .....
Kim Molvig
Thesis Supervisor

Accepted by .....
Professor Alan Henry
Chairman, Departmental Committee on Graduate Studies

LATTICE GAS HYDRODYNAMICS WITH GALILEAN INVARIANCE

by

PETER A. DONIS

S.B., Nuclear Engineering
Massachusetts Institute of Technology, 1987

Submitted to the Department of Nuclear Engineering
on September 21, 1988 in partial fulfillment of the requirements
for the Degree of Master of Science in Nuclear Engineering

\BSTRACT

A method is developed for ensuring Galilean invariance in lattice gas simulations of hydrodynamics. First the necessity for Galilean invariance is established by deriving the general hydrodynamic momentum equation for a lattice gas and showing the effects of non-Galilean invariance, specifically the need to restrict consideration to incompressible flows and the incorrect handling of passive scalars. Then a generalized formalism for describing the update rules of a lattice gas simulation is developed, in order to facilitate further work on hydrodynamic simulations using the methods discussed in this thesis. Finally, the specific lattice gas used to test the method for ensuring Galilean invariance is described and its macroscopic properties developed.

The lattice gas is realized in a FORTRAN code which is run on a MicroVAX II machine utilizing special system functions which allow direct manipulation of bits of computer storage by Boolean operations. The output of the code shows that the method used to ensure Galilean invariance which is developed can in fact meet the requirements of equilibrating to a state in which the Galilean invariance condition is met and of reaching such a state on a time scale which is short compared with the hydrodynamic time scale. The simulation is subject to statistical fluctuations, but these are expected to become smaller as the size of the simulation is increased.

Thesis Supervisor: Dr. Kim Molvig
Title: Professor of Nuclear Engineering

1 Introduction

The research described in this thesis is intended to show that a lattice gas, or cellular automaton (referred to herein by the shorthand term 'CA' for convenience) can be constructed to exhibit true Galilean invariance. This property is essential for any lattice gas which will be used to simulate hydrodynamic problems, and previous CA's, while able to exhibit Galilean invariance, have done so via methods which limited them to incompressible flows, and also introduced errors in other areas such as the transport of passive scalars. The present work will develop a method for ensuring Galilean invariance which may be applied in complete generality to hydrodynamic flows, and which preserves the correct transport laws.

The CA developed herein is not in itself capable of real hydrodynamic simulations, for reasons which will be discussed in detail in section 4 below. The Galilean invariance property is only one of several which are necessary for a hydrodynamic simulation; other such properties are discussed somewhat herein, but a fuller treatment of these other elements will be postponed to future work, since they do not bear directly on the Galilean invariance problem. The CA which ensures Galilean invariance must therefore be integrated with these other elements to produce a complete hydrodynamic simulation. In order to facilitate such integration, attention is given in section 3 to developing a general formalism for describing CA's of this class. This formalism is derived in as much generality as possible, far more than is necessary for simply studying the Galilean invariance CA discussed herein; the generality is necessary mainly because of the much-increased complexity of automata which combine all the necessary elements.

The CA is actually realized in a FORTRAN code which is run on a MicroVAX II machine, using special 'bitwise' intrinsic functions which perform Boolean operations directly on bits of computer storage. This enables the programmer to take full advantage of the Boolean nature of lattice gas simulations; these bit operations are much faster than the floating-point operations normally used in hydrodynamics codes, and they are also more 'robust' in the sense that there are no undefined operations or numerical instabilities of the type so often encountered in floating-point codes. Thus, the complexity of the code is greatly reduced, since there is no need to include error control methods, iterative evaluations of functions and derivatives, convergence criteria, and so on. The only really slow operations in the code are the input and output operations and the initialization of the variables, where floating point operations cannot be avoided.

2 Background

2.1 Galilean Invariance

In order to understand the problem which the present CA is intended to solve, it is necessary to go into some detail in discussing hydrodynamic theory, specifically to observe the effect on system behavior of not having Galilean invariance. The property of Galilean invariance itself is rather simple to grasp, and was first recognized by Galileo (hence the name); it simply means that certain physical situations are 'equivalent' in the sense that they produce the same results although they apparently involve different actions to the observer. Thus, if you are hot and want to cool down, it makes no difference whether you stand in front of a fan or get on a bicycle and ride fast enough to create a breeze; the result is the same, wind blowing past

you (from your point of view). Put this baldly, it sounds as though any reasonable simulation of a physical system would have to have this property; however, there is also a quantitative side to the question, which is not so easily dismissed as intuitively obvious.

In the just stated problem, for example, Galilean invariance would require that, if you rode your bicycle in still air at 10 miles per hour, you would experience exactly the same cooling effect as if you were standing in front of a fan which created a 10 mph wind. Mathematically, this means that the law of heat transfer from your body to the air is invariant under the transformation of coordinates

$$\begin{aligned} x' &= x - v_0 t \\ t' &= t \end{aligned} \tag{1}$$

where x, t are coordinates with reference to the ground (i.e., coordinates when standing still in front of the fan), x', t' are coordinates with reference to the moving bicycle, and v_0 is the relative velocity of the wind, in this case 10 mph. This coordinate transformation is called the Galilean transformation, and all the laws of classical (i.e., non-relativistic) physics (and hydrodynamics in particular) must be invariant under it. Of course, the full transformation involves not only the 'basic' coordinates x, t, as in equation (1), but also all other physical quantities; this will be important when we actually apply it to the hydrodynamic equations in section 2.2.

A hydrodynamic example would be the following: suppose we set up a pipe flow problem with a certain peak velocity v_0 in the center of the pipe. Then any results we obtain should be exactly the same quantitatively as those obtained from leaving the fluid within the pipe at rest and moving the pipe with velocity v_0 relative to the

fluid in the center of the pipe. The equivalence of the two cases can be given an exact mathematical expression, as will be shown in section 2.2 below; previous automata were only able to satisfy this expression by requiring incompressible flow and then re-scaling the time coordinate, which did not take care of errors in the transport of passive scalars, and was not a 'true' solution to the problem. The purpose of this thesis will be to show that the mathematical condition can be satisfied without any limitations on the scope of the simulation. In order to show this, we must derive the hydrodynamic equations for a general CA, and it is to this that we shall now turn our attention.

2.2 Hydrodynamic Equations

In order to construct the hydrodynamic equations for a CA, the primitive quantities on which all the mathematics are based must first be defined. We start out with the assumption that the lattice gas is composed of a finite number of 'types' of particles, indexed by type number j, which move about on the lattice and whose motions and interactions generate all of the properties of the system. For each particle type j, the following quantities are presumed known:

$$m_j = \text{mass of particle of type } j;$$
$$\vec{c}_{ji} = \text{velocity vector of particle of type } j;$$
$$\epsilon_j = \text{energy of particle of type } j.$$

It is important to note that these quantities can be chosen arbitrarily when constructing the CA in order to make the macroscopic equations come out in a desired form; there are no *a priori* considerations which dictate a certain choice for

any of them. For the purpose of the following derivations, however, we leave them arbitrary and make no assumptions about their nature to begin with.

The reader will notice that the velocity vector $\vec{c}_{ji}$ has a second index; this is the 'direction index' i of the velocity vector. We assume that each moving particle type has a specific number of 'directions' allowed to it, so that the full velocity vector for a given particle of that type is $\vec{c}_{ji}$, the velocity of particle type j in direction i. We assume that the magnitude of this vector is the same for all particles of type j, regardless of direction, and can be written simply as c_j. The implications of having this second index i on $\vec{c}_{ji}$ will become clear in a moment.

We can now distinguish two concepts of importance for understanding CA dynamics: 'site' and 'state'. The 'site' concept is rather obvious: the CA is constructed on a discrete lattice on which the particles move, and all such movements must be from one site to another: a particle cannot be caught 'between' sites. Time, of course, is also discrete in this model, being measured in 'steps', so that each particle must be on a lattice site at each 'step'. On a square lattice, which is used for the class of CA's to be discussed in section 3, if we define the distance between two adjacent lattice sites as 1, then all velocity magnitudes c_j must be equal to the square root of an integer (by the Pythagorean theorem). On other regular lattices, the velocity might have another form, but for all lattice gas systems velocity, like space and time, is discretized, and therefore so are the energy and momentum of the particles (if we assume that both of these must be dependent on velocity, as is usual in kinematics).

The 'state' concept is a bit different; it refers to the Boolean nature of the CA 'update rule' which determines the particle interactions. The rule cannot be Boolean

unless the input variables are Boolean, and the input variables cannot be Boolean unless the particles on the lattice obey the 'exclusion principle', so that no two can occupy the same 'state' at a given time step. The 'state', then, is defined to be the existence or non-existence of a particle of type j moving in direction i at time step t and at lattice site x, and the Boolean variable is $n_{ji}(x,t)$; it is 1 if the state is occupied by a particle, and 0 if the state is empty. Note that this means that the number of states for a given particle type j is not necessarily equal to the number of lattice sites; only for particles whose velocity $c_j = 0$ will the 'state number' be equal to the site number.

This inequality of states and sites becomes important when deriving the hydrodynamic equations for the system. The system dynamics are described most basically by a set of 'update equations', one for each species of particle present in the system; for a CA such as we have been discussing, a 'species' is defined as the ensemble of all particles of the same type j moving in the same direction i. Each update equation is then a local, discrete equation which is written as follows:

$$n_{ji}(\vec{\mathbf{x}} + \vec{\mathbf{c}}_{ji}, t+1) = n_{ji}(\vec{\mathbf{x}}, t) + \mathbf{C}_{ji} \tag{2}$$

where $n_{ji}(\vec{\mathbf{x}}, t)$ here is the Boolean variable, and the 'collision operator' $\mathbf{C}_{ji}$ contains all the information about interactions with other species of particles. The structure of $\mathbf{C}_{ji}$ is determined by the 'update rule' for the specific CA under consideration; section 3 is concerned with deriving a generalized formalism for describing CA update rules, but for this derivation we need no information about $\mathbf{C}_{ji}$'s specific structure as long as it locally conserves mass, momentum, and energy at each lattice site. The reason for this stipulation will become clear shortly.

The update equation (2) is discrete in the sense that it presupposes a specific lattice site separation and time step length, both of which are set equal to 1 for convenience and are not viewed as variables or infinitesimal quantities. We now, however, invoke the assumption that the hydrodynamic events in which we are interested happen on space scales that are much larger than the spacing between adjacent lattice sites, and on time scales which are many CA time steps long; this allows us to pass from the Boolean variables $n_{ji}(x,t)$ to the 'distribution functions' $n_{ji}(\vec{\mathbf{x}},t)$ for each species by a process of 'ensemble averaging'. This averaging process should produce a distribution function $n_{ji}(\vec{\mathbf{x}},t)$ which is normalized to 1 and which gives the probability of finding a particle of species j,i at a given lattice location $(\vec{\mathbf{x}},t)$; in order for this to be true, however, the species must be indexed by state and not by lattice site, and this explains the necessity of the second index i to distinguish between species.

The assumption underlying the ensemble averaging process is that the discrete update equation for the Boolean variable is identical in form to the continuous equation for the distribution function, because the distribution function describes the 'average behavior' of the particle species. Thus, equation (2) can be viewed not only as an equation for the Boolean variables but also as an equation for the distribution function, and with this and the time and space scaling described above, which allows us to view the CA lattice site spacing and time step length as infinitesimals, we can manipulate equation (2) as a continuous equation. We therefore move $n_{ji}(\vec{\mathbf{x}},t)$ to the left side of (2) and Taylor-expand in $\vec{\mathbf{x}}$ and t; the zero-order term of the expansion is then cancelled by the $n_{ji}(\vec{\mathbf{x}},t)$ moved from the right side, and if we

then retain only the terms to leading order, we are left with the first-order 'kinetic equation' for the continuous distribution function $n_{ji}(\vec{\mathbf{x}}, t)$:

$$\frac{\partial}{\partial t} n_{ji}(\vec{\mathbf{x}}, t) + \nabla \cdot \vec{\mathbf{c}}_{ji} n_{ji}(\vec{\mathbf{x}}, t) = \mathbf{C}_{ji}. \tag{3}$$

Note that in (3) the velocity vector $\vec{\mathbf{c}}_{ji}$ has been brought inside the derivative operator ∇; this is permissible because $\vec{\mathbf{c}}_{ji}$ is a constant and does not vary in space. This change is necessary in anticipation of taking moments over all species j, i, since the resulting equations will involve not $\vec{\mathbf{c}}_{ji}$ but $\vec{\mathbf{u}}$, which does vary in space and therefore should be included under the operator ∇.

The hydrodynamic equations, or 'Euler equations', are then obtained by multiplying the kinetic equation (3) by the primitive quantities of the system and summing over all j, i. Here the nature of the collision operator becomes important, because if it locally conserves mass, momentum, and energy, then all of the 'moments' which we take on the kinetic equation to obtain the Euler equations will vanish when taken over the collision operator. Thus, since all CA update rules with which we are concerned do in fact conserve mass, momentum, and energy locally, the right side of equation (3) vanishes when the moments are taken, and we obtain

$$\sum_{j,i} \left(\frac{\partial}{\partial t} n_{ji}(\vec{\mathbf{x}}, t) + \nabla \cdot \vec{\mathbf{c}}_{ji} n_{ji}(\vec{\mathbf{x}}, t) \right) \begin{bmatrix} m_j \\ m_j \vec{\mathbf{c}}_{ji} \\ \epsilon_j \end{bmatrix} = 0. \tag{4}$$

These moments give the Euler equations for the 'fluid variables' ρ (mass density), $\vec{\mathbf{u}}$ (average fluid velocity), and U (average fluid internal energy). We will not consider the energy equation here, which is given by the third moment in (4). The first two yield the continuity equation (mass conservation),

$$\frac{\partial}{\partial t} \rho + \nabla \cdot \rho \vec{\mathbf{u}} = 0 \tag{5}$$

and the momentum equation,

$$\frac{\partial}{\partial t}\rho\vec{\mathbf{u}} + \nabla \cdot \mathbf{P} = 0 \tag{6}$$

where the pressure tensor $\mathbf{P}$ is given by $\mathbf{P} = \sum_{j,i} m_j \vec{\mathbf{c}}_{ji}\vec{\mathbf{c}}_{ji} n_{ji}\,(\vec{\mathbf{x}}, t)$.

The pressure tensor is an important quantity in hydrodynamics, and its form makes a considerable difference in the complexity of the equations involving it, i.e., both the momentum and the energy equations. It can be shown that if the velocity vectors $\vec{\mathbf{c}}_{ji}$ satisfy certain conditions involving the tensors which can be constructed from them (which conditions will in fact be satisfied for the full hydrodynamic CA of which the present automaton is a part), then the pressure tensor $\mathbf{P}$ is isotropic and can be written as

$$\mathbf{P} = \mathbf{I}P_s + g\rho\vec{\mathbf{u}}\vec{\mathbf{u}} \tag{7}$$

where the factor g is in general not unity for lattice gases. This factor g is the crucial parameter for Galilean invariance, for if we re-write the momentum equation (6) using the pressure tensor defined in (7) and the continuity equation (5) , we obtain

$$\rho\frac{\partial}{\partial t}\vec{\mathbf{u}} + g\rho\vec{\mathbf{u}} \cdot \nabla\vec{\mathbf{u}} + \vec{\mathbf{u}}\nabla \cdot [\rho\vec{\mathbf{u}}\,(g-1)] = -\nabla P_s. \tag{8}$$

Equation (8) will only exhibit Galilean invariance if $g = 1$; this is shown by performing the coordinate transformation (1) on the equation, which is most easily done by writing the variables for the old frame in terms of the new and substituting directly into (8):

$$\vec{u} = \vec{u}' + \vec{v}_0$$

$$\frac{\partial}{\partial t} = \frac{\partial}{\partial t'} - \vec{v}_0 \cdot \nabla'$$

$$\nabla = \nabla'$$

where $\vec{v}_0$ is the relative velocity of the two frames. These substitutions yield the new equation in the primed frame:

$$\rho \frac{\partial}{\partial t'} \vec{u}' + g\rho \vec{u}' \cdot \nabla' \vec{u}' + \vec{u}' \nabla' \cdot \left[\rho \vec{u}' (g-1)\right]$$

$$+\rho (g-1) \left(\vec{v}_0 \cdot \nabla' \vec{u}' + \vec{v}_0 \nabla' \cdot \vec{u}'\right) + \left(2\vec{u}' + \vec{v}_0\right) \vec{v}_0 \cdot \nabla' \rho (g-1)$$

$$= -\nabla P_s. \tag{9}$$

Obviously, (9) is only identical in form to (8) (as it must be to satisfy Galilean invariance) if all of the 'spurious' terms involving $\vec{v}_0$ in the second line of (9) vanish, and that can only happen if $g = 1$.

The derivation of the momentum equation for non-isotropic pressure tensors is somewhat more complicated, but yields the same final condition that the factor g be unity to guarantee Galilean invariance. However, the actual mathematical form of g is slightly different from the isotropic case. This is important for the present purpose because, as will be shown in section ??, the pressure tensor for the CA developed in the present thesis is not isotropic, although the tensor for the full hydrodynamic CA to be developed using the methods discussed here will have an isotropic pressure tensor.

We should pause a moment here to reflect on the real importance of the Galilean invariance condition $g=1$; what would happen if, as has been the case for previous lattice gas simulations, g was actually a function of the mass density ρ, and not generally unity? Having done the transformation from (8) to (9), we can at once give the answer: if g is not unity, spurious effects having nothing to do with real physics will appear depending on the frame of reference of the calculation. In the pipe flow problem mentioned in section 2.1, for example, equation (8) would be valid for the case of the pipe at rest, and equation (9) for the case of the fluid initially at rest and the pipe moving with velocity $\vec{v}_0$. The simplest illustration of the difference would be that, for a given pressure drop over the pipe (i.e., right sides of both equations equal), the observed laminar flow profile would not be the same in both cases; in particular, it would not be parabolic for the case which obeyed equation (9) if it were for the case which obeyed equation (8). Other more subtle effects involve the transport of passive scalars, which would in general not be carried along at the fluid velocity $\vec{u}$. It is obvious just from the pipe flow example, however, that a simulation which does not guarantee Galilean invariance is not viable.

Previous automata got around this problem by restricting their scope to incompressible flows, which eliminates the third term in equation (8), and then re-scaling the time coordinate in order to make the Galilean transformation of the first two terms come out properly. Obviously this is restrictive since it does not allow handling of compressibility; however, there is the further problem that the handling of passive scalars is not solved by this method. The present CA labors under no such restrictions; however, before going into the specific methods used in it to ensure true Galilean invariance, it will be useful to first develop a generalized formalism

for describing the family of CA's of which the present one is a member, so that the methods used to ensure $g=1$ can be generalized. After all, the present CA, although exhibiting true Galilean invariance, is hardly suited to true hydrodynamic modeling; much more complexity and flexibility has still to be added, and the Galilean invariance solution is but a step on the road. The non-isotropy of the pressure tensor, mentioned above, is just one example of an area in which the present CA is inadequate.

3 Generalized Automaton Formalism

The purpose of this section is to construct, in as general a form as possible, an expression for a CA update rule which can be applied to any automaton used to simulate hydrodynamics. As stated in section 2.2, the 'state variables' used to describe this class of CA's are the Boolean variables $n_{ji}(x,t)$ which indicate the presence or absence of a particle of type j moving in direction i as a function of the time and space coordinates. The lattice used is a square lattice, so that the magnitude of each velocity vector c_j must be the square root of an integer; however, no further restriction will be made in this section on the velocity vectors for the particle types.

Each particle type j also has associated with it three other quantities, the 'primitive variables' already referred to in connection with the derivation of the hydrodynamic equations. Here, however, we do not leave these variables totally arbitrary, but impose definite conditions on their form and values. To be specific, the class of CA's which we will now consider has the following restrictions on the primitive variables:

$$\text{mass } m_j \text{ either 0, or nonzero value indep. of } j;$$
$$\text{momentum } p_j = m_j c_j \text{ for } m_j \neq 0 \text{ or } c_j \text{ for } m_j = 0;$$
$$\text{energy } \epsilon_j = \tfrac{1}{2} m_j c_j^2 \text{ for } m_j \neq 0 \text{ or } p_j \text{ for } m_j = 0.$$

The reader will notice the restriction on the mass m_j; all particle types j with $m_j \neq 0$ must have the same value for their mass (usually taken to be 1 for simplicity, but may be given another value if convenient). This restriction is mainly to reduce the number of possible interactions which satisfy the local conservation laws of mass, momentum, and energy; if varying masses are allowed on the lattice the mathematics becomes overly complicated without introducing any really new physics.

One feature of this class of CA's which immediately distinguishes it from previous automata is the presence of the energy variable ϵ_j as a property distinct from the mass m_j. The fact that different particle types with different energies and energy/mass ratios can exist on the lattice means that the system as a whole can exhibit a property *temperature* which can vary with time and can be shown to follow the familiar laws of thermodynamics (this subject is dealt with in reference [?], a previous thesis by the present author, and also in reference [?]). In hydrodynamic terms, this means that the energy equation, derived from the third moment of the kinetic equation as shown in (4), is not simply equivalent to the continuity equation derived from the first moment, but brings in new and important physics of its own. Thus, this class of CA's requires all three Euler equations to completely describe its dynamics, although here we are concerned only with one of them.

Our task now is to construct an 'update rule' for each Boolean variable $n_{ji}(x,t)$; that is, we want a finite set of Boolean operations which we can perform on $n_{ji}(x,t)$ to obtain $n_{ji}(x,t+1)$. The series of operations need not be identical for all j and i;

however, the form in which we derive the update rule should, for convenience, be as independent of j and i as possible. This is because each of these update rules will give the collision operator in the update equation (2), and we would like to preserve as much as possible the generality of form of that equation.

As it happens, one such form which fits these requirements admirably is that which has already been used to describe the update rule for the temperature-modeling CA of the author's previous thesis. In Boolean notation, this rule has a very simple and intuitive form:

$$n_{ji}(x, t+1) = \left[n_{ji}(x,t) \wedge \overline{\mathcal{A}_{ji}}\right] \vee \mathcal{C}_{ji} \tag{10}$$

where $\mathcal{C}_{ji}$ and $\mathcal{A}_{ji}$ are the 'creation sum' and 'annihilation sum' respectively, and include the effects of all interactions which can either create or destroy a particle of type j in direction i. The rule (10) then says simply that a particle of type j in direction i will be present at time $t+1$ if: a) it was present at time t and was not destroyed by, any process, or b) it was created at time t by some process. When averaged as described in section 2.2, this equation assumes the simple form

$$\frac{\partial}{\partial t} n_{ji}(\vec{\mathbf{x}}, t) + \nabla \cdot \vec{\mathbf{c}}_{ji} n_{ji}(\vec{\mathbf{x}}, t) = \mathbf{C}_{ji} = C_{ji} - A_{ji} \tag{11}$$

where C_{ji} and A_{ji} are the 'averaged' $\mathcal{C}_{ji}$ and $\mathcal{A}_{ji}$ and represent the probability that a particle at location $(\vec{\mathbf{x}}, t$ will be created or annihilated, respectively.

We should note that this update rule is entirely *local* (it only involves functions at lattice site x) as long as $\mathcal{C}_{ji}$ and $\mathcal{A}_{ji}$ can be constructed entirely from Boolean variables at site x. Since we will observe this condition, we can suppress the space variable in our notation and simply write n_{ji}^{t} and n_{ji}^{t+1} for our state variables, in the interest of compactness. It will also be convenient to replace the Boolean notation

of (10) by conventional arithmetic notation, in order to facilitate the change to continuous variables which must be made in order to derive hydrodynamic equations as was done in section 2.2 above. Thus, the Boolean functions AND ($\wedge$) and OR ($\vee$) will be represented by multiplication and addition in what follows; the reader should bear in mind that all of the equations in this section are Boolean equations and all variables and functions can only assume the values 1 and 0.

3.1 Truth Functions

In order to derive the forms of the creation and annihilation sums $\boldsymbol{C}_{ji}$ and $\boldsymbol{A}_{ji}$ all possible particle interactions on the lattice must be somehow described. In work dealing with the temperature-modeling CA discussed in the author's previous thesis, processes were indexed as C_{ji}^{p} or A_{ji}^{p}, where p was the 'process index' and the C or A meant that the process created or annihilated a particle of type j in direction i. However, this notation did not take advantage of the inherent symmetries of the processes which were being described, so that the same process could be described by two or three equally valid notations, and processes which were related by symmetry operations did not bear any simple resemblance in notation. In the present formalism processes are instead described by 'truth functions' T_{i}^{p}, indexed only by the 'process index' p and a 'direction index' i, which allows us to distinguish rotations of a given process p, which involve the same particle types but different directions. It is not necessary to add a particle index j; such an index would give no new information, and could not take the place of either of the two indices already described, since a given process p must always involve the same particle types, even though it can be rotated to involve different directions.

Figure 1: **Odd Photon Process**

The usefulness of the truth function notation stems from the fact that processes on the square lattice exist in pairs which are related by the TP operator, which stands for time reversal and parity reversal (reflection through the origin). Thus, each process index p and direction i actually indexes two interactions, one 'forward' and the other 'inverse', which can be transformed into each other by the TP operator. An example of such a pair is given in Figure 1; we will use the Feynman diagram notation to diagram interactions, so that the T operator simply reverses the direction of each arrow and the P operator reflects each arrow through the center of the diagram. Here the straight lines are particles with $m_j = 1$ and the wiggly lines are 'photons', or particles with $m_j = 0$. Although in this diagram only two types of velocity vectors are shown, the TP-symmetry pairs exist for other velocity vectors and particle types as well.

Each pair of processes is then described by the pair of truth functions T_i^p, $\overset{*}{T}{}_i^p$. The notation $\overset{*}{T}{}_i^p$ stems from the fact that the two functions are 'dual' to each other; the TP operator translated into Boolean terms means simply replacing each Boolean variable n_{ji}^t by its complement $\overline{n}_{ji}^t$, which is the dual operation (note that the Boolean

notation for the complement has been retained). The Boolean variables which make up the truth function are those for the states which the process involves: the function is the product of the variables for all the 'input states' and the negations of all the variables for the 'output states', and therefore expresses the condition that all the necessary input states be occupied and all the necessary output states be empty for the process to occur. Since the TP operator reverses input and output states, the Boolean dual operation should indeed, by this reasoning, produce $\overset{*}{T}{}_i^p$ from T_i^p (since the complement of the complement gives the original Boolean variable). In this notation, the truth functions for the processes shown in Figure 1 are:

$$T_i^1 = n_{1,i}^t \, n_{p,i+1}^t \, \overline{n}_{2,i}^t \tag{12}$$

$$\overset{*}{T}{}_i^1 = \overline{n}_{1,i}^t \, \overline{n}_{p,i+1}^t \, n_{2,i}^t \tag{13}$$

where $j = 1$ denotes the particles moving horizontally and vertically, $j = 2$ denotes the particles moving 'diagonally', and $j = p$ denotes the photons. The direction indices i run from 1 to 4 for this specific system, being indexed clockwise starting with 'straight up' for $j = 1, p$ and with 'up and to the right' for $j = 2$. The 'dual' nature of the pairing should be obvious from (18) and (??).

The actual number of process indices p, particle types j, directions i, and so on depends, of course, on the specific CA; however, in this formalism we are making no assumptions about these matters. The only point which is crucial is that the truth functions for each possible process pair be producible given the primitive variables m_j, c_j, ϵ_j for the system. This condition is met by considering the main requirement which all possible processes must meet: they must conserve mass, momentum, and

energy locally at the lattice site at which they take place. Using these conservation laws, the possible combinations of states can be systematically examined, and 'interactions' which do not obey the laws can be discarded, leaving only the permissible processes. Once the involved states for each process are known, the forward and inverse truth functions can easily be constructed. The decision as to which process of a pair is 'forward' and which 'inverse', or how to assign the indices p, j, i, is completely arbitrary, although it is advisable to choose such assignments to be as straightforward mnemonically as possible.

3.2 Process Masks

When processes are enumerated and examined for a CA, it may turn out that some process has only one output state (i.e., it only creates one particle). If that is the case, then the inverse process is 'spontaneous'; that is, having only one input state (since input and output states are reversed to obtain the inverse), it can occur without the necessity of a collision between two or more particles. These spontaneous processes have associated with them a 'decay coefficient' ν_p, which is a number between 0 and 1 that determines the probability of the spontaneous process occurring. Letting this coefficient in general be other than unity allows one to have considerable control over the macroscopic properties of the system, as will be shown in section 4.3 below.

There are many ways of implementing this coefficient ν_p into the update rule. One method is to multiply the truth function for each spontaneous process by ν_p; while straightforward and simple, this has the fatal flaw of removing the Boolean character of the truth function, and therefore of the update rule, so that another method must be sought. The solution arises from the fact that ν_p is an 'average'

or 'probabilistic' quantity; that is, it only strictly applies when a large number of time steps and space locations are considered. We can therefore use any method in the update rule which will yield the correct average over a large number of process 'events', no matter what its nature microscopically; we need not worry about trying to locally simulate (using a random number generator or some similar method) the probabilistic nature of the overall interaction. The method which is assumed in this general formalism is that of creating a Boolean variable P^p, or 'process mask', which is 1 on a fraction ν_p of the total number of lattice sites. Thus, over a large enough number of time steps, the decay process p will happen a fraction ν_p of the time. In order to keep the formalism consistent, we also define process masks for non-spontaneous processes; we simply set them to 1 everywhere, so that in effect we are associating a coefficient $\nu_p = 1$ with these processes (they happen whenever their truth function is 'on').

Another function of the process mask once defined, which was quite useful in the CA described in the present author's previous thesis, is to take care of cases in which processes 'conflict' (i.e., share input or output states). Specifically, it may happen that several different processes produce only one output state, which happens to be the same one; then the inverse process, with the same input state in both cases, will have several possible output states. Again, a random number generator could be used to choose between the possible states at each occurrence of the decay; however, the process masks afford a much faster solution. If the number of processes p which share states in this way is N, then we simply 'seed' each mask with a probability of ν_p/N, taking steps to ensure that no two masks are 1 at the same site.

The total 'truth condition' for a process p to occur is then given by $P^pT_i^p$; the truth function and the process mask must both be 1 at the lattice site for the process to occur. From here on in this discussion, therefore, the terms 'truth function' and 'truth condition' will be assumed to stand for $P^pT_i^p$ unless the context makes it clear that only T_i^p is meant.

3.3 Matrix Element

The truth condition $P^pT_i^p$ is the primary condition for a process to occur; however, it has already been mentioned that several processes may share input or output states and therefore 'compete'. The use of process masks can help to alleviate this problem by effectively 'labeling' sites so that only certain processes can happen at each; however, this method can quickly become cumbersome when a large number of interactions are possible on the lattice, since steps must be taken to ensure that the masks for two conflicting processes are not both 1 at a given site. Therefore, it is desirable to find some other method of dealing with competing processes.

The simplest method, and the one which will be treated in this formalism, is to require that the truth functions for all competing processes be 'off' in order for a process whose truth condition is 'on' at a given site to actually happen. We therefore define another Boolean function called the 'matrix element' M (whose indices have not yet been determined) which is 1 only if no other competing truth functions are 1 at the site in question. To keep the formalism general, however, we cannot assume anything about the nature of the processes which would tell us which p indices 'compete' with one another. Therefore, the matrix element M must include *all* other processes, whether they actually 'compete' or not, in the general formalism. This condition is evidently stricter than necessary to avoid competition,

since by it two processes p which do not share either input or output states will still be mutually excluded by the matrix element. The advantages and drawbacks of this method will be discussed in a moment; first, however, let us develop the form of the matrix element based on this criterion.

Logically, the assertion that no competing truth functions be 'on' is expressed by multiplying (linking with AND) the negations of all the truth functions. We might suppose that simply negating all processes with a p index different from that of the process we are interested in would be enough; however, to be completely general, we should include both indices on the truth function, p and i, in order to take into account processes with the same p but different i which might also compete. We can therefore state the form of the general matrix element M_i^p:

$$M_i^p = \prod_{p' \neq p} \prod_{i'} \overline{\left(P^{p'} T_{i'}^{p'}\right)} \overline{\left(\overset{*}{P}^{p'} \overset{*}{T}_{i'}^{p'}\right)} \prod_{i' \neq i} \overline{\left(P^{p} T_{i'}^{p}\right)} \overline{\left(\overset{*}{P}^{p} \overset{*}{T}_{i'}^{p}\right)} \tag{14}$$

The *full* condition for process p, i to happen is now given by $M_i^p P^p T_i^p$; the process will actually take place only if: a) the truth condition is satisfied (process mask and truth fn. both 1), and b) the truth conditions for *all* other processes are *not* satisfied. For convenience in further discussion (and also for easier code implementation) we transform (14) by De Morgan's Rule into

$$M_i^p = \overline{\left(\sum_{p' \neq p} \sum_{i'} \left[P^{p'} T_{i'}^{p'} + \overset{*}{P}^{p'} \overset{*}{T}_{i'}^{p'} \right] + \sum_{i' \neq i} \left[P^{p} T_{i'}^{p} + \overset{*}{P}^{p} \overset{*}{T}_{i'}^{p} \right] \right)} \tag{15}$$

which becomes our general matrix element form.

It is fairly easy to see that the 'inverse matrix element', $\overset{*}{M}_i^p$, is equal to M_i^p: we simply replace each Boolean function in either (14) or (15) by its inverse or 'dual', and remember that the dual of the dual is equal to the identity operator. This is

easiest to see in (15), where the substitution immediately leads to the same equation. Thus, we can use the same matrix element for both the forward and inverse process for each p and i. This equivalence, although not further discussed in this thesis, is important in thermodynamic theory for reasons which are discussed in reference [?].

Although the matrix element (15) is attractive from the standpoint of simplicity, it has, as already mentioned, some drawbacks from the standpoint of actual physical modeling. The most obvious drawback is that interactions on the lattice happen less often than they would if only 'competing' processes were disallowed and processes which shared no input and output states were allowed to take place simultaneously at a given lattice site. This effect is negligible at low densities where the probability distributions $n_{ji}(\vec{\mathbf{x}}, t)$ are all much less than 1; however, at higher densities, the probability of two truth functions being satisfied at the same lattice site gets larger, and we will find that the automaton becomes 'frozen' as the matrix element disallows more and more processes, even if they do not actually compete. However, there is no immediately apparent way to formalize generally the conditions that would have to be satisfied for two truth conditions $P^p T_i^p$, $P^{p'} T_{i'}^{p'}$ to share input and/or output states, since this depends on the specific forms of the truth functions, and therefore on the specific structure of a given CA.

There is also the possibility of dispensing with the matrix element altogether by using a 'cycle rule', which updates the processes one at a time according to some definite order, so that conflicts never arise. Thus, supposing that we have three processes p and four directions i for each process, we would take a state $n_{ji}(x, t)$, update it using process $p = 1, i = 1$, say, then take the output from that update and update it using $p = 1, i = 2$, and so on until we had covered all possible

combinations of p, i. We could even vary the order in which the processes are taken on different time steps, in order to achieve some sort of 'equal mixing' of the effects of each process. It is not clear at this writing whether such a 'cycle rule' could be formalized in a form sufficiently simple as to allow expression in the form (10); certainly it is not clear *a priori* that the form (10) is valid for the cycle rule, since each process, whether creating or annihilating a particle j, i, acts on a different 'input state' than all other processes, so that they cannot be summed together in any simple way.

Future work on the complete hydrodynamic CA may attempt to use the 'cycle rule', or some other method of avoiding the 'freezing out' of processes at high densities that occurs with the present matrix element. However, this problem is not directly related to the Galilean invariance question, so we will devote no further attention to it in this thesis. The simulations discussed in section 5 were for the most part done at densities low enough that the matrix element effects were negligible.

3.4 Creation and Annihilation Sums

We are now in a position to construct the creation and annihilation sums $\boldsymbol{C}_{ji}$ and $\boldsymbol{A}_{ji}$. First, we must define two Boolean functions which will enable us to write the two sums in a very symmetric and elegant manner. The first is the 'sign function' S_j^p; it is defined arbitrarily to be 1 if the 'forward' process p creates a particle of type j (irrespective of direction). Therefore, the complement $\bar{S}_j^p$ is 1 if the 'forward' process p annihilates a particle of type j, or, equivalently, if the inverse process p creates a particle of type j. The sign function may therefore be viewed as implicitly defining which process of each dual pair is 'forward' and which 'inverse'.

The second function which we shall define is the 'index function' $\mathcal{F}^p_{j,i,i'}$. The plethora of indices here is not really that difficult to understand; the index function will be 1 if the permutation of indices p, j, i, i' is 'allowed', in the sense that the variable n_{ji} is an 'argument' of the truth function pair $T^p_{i'}$, $\overset{*}{T}{}^p_{i'}$. A Boolean variable is said to be an 'argument' of a truth function pair if it appears in one function of the pair and its complement appears in the other (it makes no difference which fn., forward or inverse, has the variable and which has the complement). The function $\mathcal{F}^p_{j,i,i'}$ is therefore something of a 'bookkeeping' function which matches each truth function pair with the variables n_{ji} which it is capable of creating or annihilating. The indices j, i must of course refer to the indices of the creation and annihilation sums, which is why it is necessary to introduce a second direction index i' for the truth function pair (since its direction index need not, in general, match the direction indices of all its arguments).

Armed with these two functions, we can now state the forms of the creation and annihilation sums:

$$\mathcal{C}_{ji} = \sum_p \sum_{i'} \mathcal{F}^p_{j,i,i'} \left[S^p_j M^p_{i'} P^p T^p_{i'} + \overline{S}^p_j M^p_{i'} \overset{*}{P}{}^p \overset{*}{T}{}^p_{i'} \right] \tag{16}$$

$$\mathcal{A}_{ji} = \sum_p \sum_{i'} \mathcal{F}^p_{j,i,i'} \left[\overline{S}^p_j M^p_{i'} P^p T^p_{i'} + S^p_j M^p_{i'} \overset{*}{P}{}^p \overset{*}{T}{}^p_{i'} \right] \tag{17}$$

The symmetry of (16) and (17) will become quite useful when the update rule is averaged and converted to continuous variables and cast in a form like (11). This is not actually done here, however, since it will not be necessary to derive the Galilean invariance condition. This completes the development of the generalized automaton formalism; the update rule (10), combined with the sum definitions (16) and (17) and the matrix element (15), can be applied to any CA of the general class defined at the beginning of this section.

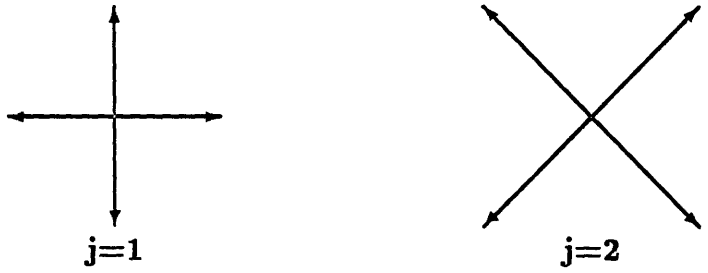

Figure 2: **012 CA Particles and Directions**

4 The Galilean Invariance Automaton

Having discussed at length the theoretical background for the Galilean invariance problem, it is now time to describe the specific CA which is used to solve it. The generalized formalism developed in the previous section will be used to describe the update rule; however, now the specific forms of the Boolean functions and the limits on the indices can be given. First, however, we must specify the 'primitive variables' for this automaton. There are three types of particles on the lattice, indexed by $j = 0, 1, 2$; the j indices are set up so that $c_j = \sqrt{j}$. Thus, the $j = 0$ particle, or 'stopped particle', does not move but always remains at the same lattice site until annihilated by some process. The $j = 1$ particle moves horizontally and vertically, and the $j = 2$ particle moves diagonally, as shown in figure 2. This range of j indices gives rise to the name '012 CA' for the automaton, which will be used hereafter to refer to it.

All particles have mass $m_j = 1$; there are no massless 'photons' in the 012 CA. The velocity vectors have already been given, and the energies are $\epsilon_j = \frac{1}{2} m_j c_j^2 = \frac{1}{2} j$. The moving particle types each have four possible direction indices, as shown in

Figure 2, in the range $i = 0$ to $i = 3$. This completes the listing of the primitive variables for the 012 CA, and we now proceed to derive the update rule according to the generalized formalism of section 3.

4.1 Update Rule

An examination of the range of possible input and output states for the 012 CA will soon lead to the conclusion that only three pairs of processes are possible which locally conserve mass, momentum, and energy. These are all diagrammed in Figure 3, and the p indices and definitions of forward and inverse for each process pair are as shown there. In each diagram in Figure 3, the processes have been shown with direction index $i = 0$; rotation of each diagram clockwise through 90 degrees produces, in successive steps, the $i = 1$, $i = 2$, and $i = 3$ diagrams for each process pair. The reader will note several symmetries in the 'self-collision' processes ($p = 2, 3$) which will be discussed in a moment.

From the diagrams in Figure 3 it is a simple matter to construct the truth functions for the three process types. We need only do so for the forward processes, since the dual operation will give us the inverses immediately once we have the forward truth functions. We adopt for convenience the notation $n_{0,0}$ for the 'stopped' particle; since it does not move, it really has no 'direction index', but in order to keep the formalism consistent we assign all stopped particles the index $i = 0$, and construct the update rule accordingly. The three forward truth functions then are:

$$T_i^1 = n_{1,i}\, n_{1,i+1}\, \overline{n}_{2,i}\, \overline{n}_{0,0} \tag{18}$$

$$T_i^2 = n_{1,i}\, n_{1,i+2}\, \overline{n}_{1,i+1}\, \overline{n}_{1,i+3} \tag{19}$$

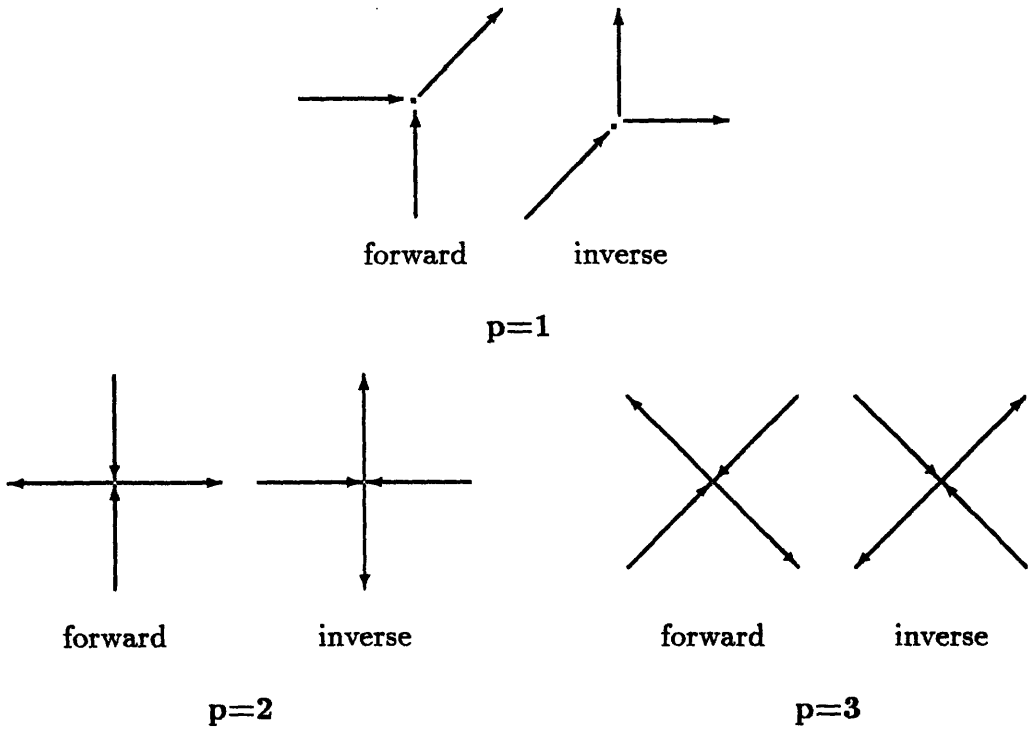

Figure 3: **Process Diagrams for 012 CA**

$$T_i^3 = n_{2,i}\, n_{2,i+2}\, \overline{n}_{2,i+1}\, \overline{n}_{2,i+3} \tag{20}$$

where we have suppressed the time index t since it is understood that all variables are evaluated at the same time step.

The reader will perhaps have noticed at this point that of the three process pairs described in (18), (19), (20), *none* are 'spontaneous' in the sense defined in section 3.2. It would seem, then, that process masks for the 012 CA are superfluous, since they would all be 1 in any case. However, it is desirable for reasons of flexibility to attach a coefficient ν_p that is different from 1 to at one process, even if it is not strictly 'spontaneous'. Such a process would have to be one in which two or more types of particles are involved, since the flexibility desired is in controlling the

thermal equilibration of the system (see section 4.3 for a fuller explanation of this), and the only such process in the 012 CA is the $p = 1$ process. (The 'self-collision' processes $p = 2, 3$ in this model must have $\nu =$ because of symmetry properties such as those discussed in Appendix B as well as the above reasons.) We therefore choose to define a coefficient ν_1 for the inverse $p = 1$ process which is in general not unity; therefore, although all other process masks are 1, the mask $\overset{*}{P}{}^1$ is 1 only on a fraction ν_p of the lattice sites.

Once we know the truth functions and process masks, the rest of the update rule then follows from the general equation (10), the forms (16) and (17) for $\mathcal{C}_{ji}$ and $\mathcal{A}_{ji}$, and the matrix element (15). The only two functions not yet known are the sign function S_j^p and the index function $\mathcal{F}^p_{j,i,i'}$. Inspection of the truth functions (18), (19), (20) will show that the sign function must be defined as follows:

$$S_j^p = \begin{cases} 1 & \text{for } p = 1 \text{ and } j = 0, j = 2; \\ 0 & \text{for all other cases.} \end{cases} \tag{21}$$

The index function is a bit more complex, and its exact nature is not really germane to the discussion here, as long as it is known for all permutations of its indices. Discussion of the detailed structure of $\mathcal{F}^p_{j,i,i'}$ will therefore not be given here, but can be found in Appendix B. A few features can, of course, be easily gleaned from the truth functions: $\mathcal{F}^p_{j,i,i'}$ must be 0 for $j = 0$ and $i \neq 0$ (regardless of p and i'), and must also be zero for $j = 2$ and $i' \neq i$, since the direction index of the type 2 particle always matches that of the truth function. Finally, the index function serves to 'pick out' the correct self-collision permutation of p, j; thus it is 0 for $p = 2, j \neq 1$ and $p = 3, j \neq 2$. The rest of the structure can be derived in similar fashion, so that we can assume that $\mathcal{F}^p_{j,i,i'}$ is a known function; its exact values will not matter in the development to follow.

The upshot of all this is that, using the generalized formalism of section 3 and the truth functions and primitive variables defined above, a complete Boolean update rule can be constructed for the 012 CA, and translated into computer code for implementation. The rest of this section will therefore be concerned with the macroscopic behavior of the 012 CA, assuming that the microscopic behavior is exactly known.

4.2 Equation for Galilean Invariance

Having defined the 'primitive variables' for the 012 CA, we must now use them to derive forms for the macroscopic variables which appear in the hydrodynamic equations, specifically the momentum equation (6), in order to derive an expression for the factor g. Thus, we need expressions for the mass density ρ, the fluid velocity $\vec{\mathbf{u}}$, and the pressure tensor defined by $\mathbf{P} = \sum_{j,i} m_j \vec{\mathbf{c}}_{ji} \vec{\mathbf{c}}_{ji} n_{ji}(\vec{\mathbf{x}}, t)$. These will in turn lead us to an expression for g, which must be 1 to guarantee Galilean invariance.

We must first define the probability distributions $n_{ji}(\vec{\mathbf{x}}, t)$ for the three types of particles present on the lattice. Letting S be the number of lattice sites, we have S possible 'states' for the stopped particles and $4S$ states for each type of moving particle. If $N_j(t)$ denotes the number of particles of type j present on the lattice at time step t, then the probability of finding a particle of type j in a given state is simply the number of particles divided by the total number of states. We will denote this 'averaged' probability by n_j; the three n_j are then given by

$$n_0 = \frac{N_0}{S}$$

$$n_1 = \frac{N_1}{4S}$$

$$n_2 = \frac{N_2}{4S} \tag{22}$$

The averaged probabilities n_j are then, on the large hydrodynamic scale, taken to be the values of the distribution function (or, more accurately, expectation value) $n_{ji}(\vec{\mathbf{x}}, t)$ at the spatial location $\vec{\mathbf{x}}$ defined by the 'lattice cell' over which the averages n_j are taken. This is justified by the 'continuum limit' arguments given in section 2.2.

The mass and energy densities are now easily generated; we define

$$\rho \equiv \frac{1}{S}\sum_j N_j, \tag{23}$$

$$\epsilon \equiv \frac{1}{2S}\sum_j N_j \epsilon_j, \tag{24}$$

and obtain immediately

$$\rho = n_0 + 4n_1 + 4n_2, \tag{25}$$

$$\epsilon = n_1 + 2n_2. \tag{26}$$

The difference in normalization between ρ and ϵ will become convenient when the g expression is derived below. The energy density ϵ, while not directly present in the momentum equation, is necessary for calculating the thermodynamic equilibrium of the system (as is done in section 4.3 below), and also will arise in the evaluation of the pressure tensor.

We will not derive the form for the fluid velocity, since its exact form will not affect the g equation, and therfore come to the pressure tensor. Using the form $\mathbf{P} = \sum_{j,i} m_j n_j \vec{\mathbf{c}}_{ji}\vec{\mathbf{c}}_{ji}$, we find that

$$\mathbf{P} = \mathbf{I}\left[\sum_j 2c_j^2 n_j \left(1 - \frac{1}{2}\beta^2 u^2 f_j'' \frac{a_j}{\overline{n}_j - n_j}\right)\right] + \frac{1}{2}\beta^2 \vec{\mathbf{u}}\cdot\mathbf{T}\cdot\vec{\mathbf{u}} \tag{27}$$

where the coefficients a_j and f_j'' depend in a complex fashion which we shall not discuss on the exact distribution function of particles in the lattice cell; the coefficient β is given by $\beta = \rho/2\epsilon$ and comes from the coefficient of the first-order term of the expansion of the distribution $n_{ji}(\vec{\mathbf{x}}, t)$ about its average value n_j, and we have introduced the second-rank identity tensor **I** and the fourth-rank tensor

$$\mathbf{T} = \sum_{j,i} n_j f_j'' \vec{\mathbf{c}}_{ji} \vec{\mathbf{c}}_{ji} \vec{\mathbf{c}}_{ji} \vec{\mathbf{c}}_{ji} \tag{28}$$

Also, in (27) we have retained the notation $\overline{n}_j$ to represent $1 - n_j$ for compactness.

The pressure tensor (??) is not isotropic, as will be seen in a moment, for the 012 CA presently under discussion, since the fourth-rank tensor **T** is not completely symmetric. It is worth noting, however, that for the complete hydrodynamics code which will draw on the work discussed here, the tensor **T** is in fact completely symmetric, so that the pressure tensor for that system is isotropic. Non-isotropy of the pressure tensor is unimportant for the present CA, since it is not intended to model actual fluid flows but only to investigate methods of ensuring that $g = 1$; for a simulation of real flows, however, isotropy of **P** is highly desirable because of the much simpler form of the Euler equations, as shown in section 2.2.

The coefficient of the identity tensor **I** in (27) is the 'scalar pressure' P_s, which we are not concerned with here because it does not affect the factor g; we therefore need evaluate only the last term in (27) to obtain an expression for g. Furthermore, for the purposes of the present discussion we will take only the low-density limit of g, meaning that we can set $f_j'' = 1$ and drop it out of the equation, and that we assume that all complements $\overline{n}_j$ are 1 also. This yields the expression

$$\mathbf{T} = 4n_2\Delta^{(4)} + \delta^{(4)}(2n_1 - 8n_2) \tag{29}$$

where $\Delta^{(4)}$ is the completely symmetric fourth-rank tensor and $\delta^{(4)}$ is the 4-index Kronecker delta (which is 1 only if all indices are equal). Performing the double dot product in the last term of (??) then yields a term of the form

$$4\beta^2 n_2 \vec{u}\vec{u} \equiv g\rho\vec{u}\vec{u} \tag{30}$$

which yields the expression for g when $\beta = \rho/2\epsilon$ is substituted:

$$g = \frac{\rho n_2}{\epsilon^2} \tag{31}$$

We can finally render this in terms of the known occupation numbers n_j in order to have an equation directly derivable from the automaton output:

$$g = \frac{n_2\left(n_0 + 4n_1 + 4n_2\right)}{\left(n_1 + 2n_2\right)^2} \tag{32}$$

Our condition for Galilean invariance is now simply $g = 1$, or, manipulating equation (32),

$$\left(n_1 + 2n_2\right)^2 = n_2\left(n_0 + 4n_1 + 4n_2\right)$$

$$n_1^2 + 4n_1 n_2 + 4n_2^2 = n_0 n_2 + 4n_1 n_2 + 4n_2^2$$

$$\Rightarrow n_1^2 = n_0 n_2. \tag{33}$$

Equation (33) is quite simple and elegant, and its simple form depends crucially on the mass and energy density normalizations chosen in (23) and (24) being different. In order to actually calculate g in the computer code, equation (32) is used, since g may not be exactly 1; however, equation (33) is more useful intuitively in grasping the nature of the $g = 1$ condition.

4.3 Equilibrium Properties

We will now study in some detail the equilibrium state of the 012 CA system under the constraint $g = 1$; our aim is to show that the equilibrium occupation numbers n_j can be expressed in terms of the fluid variables ρ and ϵ. Such an expression is useful for two reasons. First, it enables us to put a 'check' on the computer code by comparing the actual occupation numbers with those predicted on the basis of the ρ and ϵ in the lattice cell. Second, it is important for any further development of hydrodynamics, transport theory, and other macroscopic properties of the system to understand the equilibrium, the degrees of freedom inherent in the system, and so on.

We start with equations (25) and (26) for the mass and energy density, equation (33) for the $g = 1$ constraint, and the thermodynamic equilibrium condition for the 012 CA

$$n_1^2 \overline{n}_0 \overline{n}_2 = \nu n_0 n_2 \overline{n}_1^2 \tag{34}$$

where the 'average decay coefficient' ν is equal to the decay coefficient ν_1 for the inverse $p = 1$ process that was introduced in section 4.1. We can immediately combine equation (34) with (33) to cancel the occupation numbers on both sides of (34), leaving only the complements:

$$\overline{n}_0 \overline{n}_2 = \nu \overline{n}_1^2 \tag{35}$$

which will determine the equilibrium ν given the three occupation numbers n_j; thus we can see that the decay coefficient also will be determined from the fluid variables.

We next observe that equation (33) can be recast in the form

$$\frac{n_0}{n_1} = \frac{n_1}{n_2} \tag{36}$$

so that we can define the ratio $R = n_0/n_1 = n_1/n_2$, and write equations (25) and (26) as follows:

$$\rho = n_2 \left(R^2 + 4R + 4\right) = (R+2)^2 n_2 \tag{37}$$

$$\epsilon = (R+2)\, n_2. \tag{38}$$

Equations (37) and (38) at once yield the solutions for R and n_2:

$$R = \frac{\rho}{\epsilon} - 2 \tag{39}$$

$$n_2 = \frac{\epsilon^2}{\rho} \tag{40}$$

and equation (35) yields the solution for ν:

$$\nu = 1 - \frac{(1-R)^2 n_2}{(1 - Rn_2)^2}. \tag{41}$$

These three equations completely determine the occupation numbers and ν in terms of the fluid variables ρ and ϵ.

In order to understand exactly what we have just done, let us now define the thermodynamic temperature of the system. We first make the notational definition $\hat{n}_j \equiv n_j/\overline{n}_j$; then we define the temperature as

$$T \equiv \frac{1}{\log\left(\frac{\nu\hat{n}_0}{\hat{n}_1}\right)}. \tag{42}$$

At thermodynamic equilibrium, from equation (34) we obtain the condition that

$$\frac{\nu\hat{n}_0}{\hat{n}_1} = \frac{\hat{n}_1}{\hat{n}_2}, \tag{43}$$

and if we define a 'particle temperature' T_{12} as

$$T_{12} \equiv \frac{1}{\log\left(\frac{\hat{n}_1}{\hat{n}_2}\right)}, \tag{44}$$

then it is obvious that at equilibrium $T_{12} = T$. More importantly, however, the equilibrium temperature T is exactly determined by the fluid variables ρ and ϵ and the $g = 1$ condition, since they determine the occupation numbers and ν through equations (39), (40), and (41). Thus, what we have done is to show that the thermodynamic temperature is not really a free variable of the system, if ρ and ϵ are considered free variables.

Another implication of what we have done is that the system must be allowed to relax to $g = 1$; it *cannot* be constantly maintained in a $g = 1$ state. To show this, recall that we defined the ratio R between the occupation numbers, and that this definition did *not* depend on the system being in thermodynamic equilibrium. In fact, our solutions for R and n_2 were actually solutions for the condition $g = 1$ *only*, without reference to equilibrium at all! The only place in which we used the equilibrium condition was in determining ν from the other variables. The reader might at this point accuse the author of a bit of false advertising, since this section was entitled 'Equilibrium Properties' and it seems as though the system properties can be determined out of equilibrium as well; however, consider what adding the equilibrium condition does to the system behavior. We can determine the necessary occupation numbers for $g = 1$ without using any equilibrium condition, and these occupation numbers are uniquely defined in terms of the fluid variables ρ and ϵ; however, the system will only reach those occupation numbers in steady state if the decay coefficient ν is the correct value to make those occupation numbers define a thermodynamic equilibrium state. If ν is not at the proper value, given by equation (41), then the system will *not* equilibrate to the proper occupation numbers for g to equal 1. And, even if ν is at its proper value, the system need not start out in

thermodynamic equilibrium; if it is perturbed from that equilibrium, it will take time to relax again, and during that time g will not be 1.

Therefore, the problem of ensuring $g = 1$ boils down to the problem of ensuring that ν is at the correct value given by equation (41); if that is the case, then given enough time the system will equilibrate and reach a steady state in which the occupation numbers are in the correct ratios R to give $g = 1$. In order for this to be satisfactory for hydrodynamic simulations, the equilibration time for the 'lattice cell' must be very short compared to the hydrodynamic time scale, so that macroscopically the system is perceived as having $g = 1$ all the time, because the small perturbations and re-relaxations happen too fast to be distinguished. These two requirements–make ν assume the correct value, and make sure that the system relaxes to $g = 1$ fast enough–form the basis for judging the code results given in section 5; in the remainder of this section we will discuss the method used to meet these requirements in the 012 CA.

4.4 Methods For Ensuring G = 1

Since the problem of ensuring $g = 1$ boils down to ensuring that ν is the correct value, and since the correct value for ν is given by equation (41), the simplest solution would seem to be determining ρ and ϵ at each time step, calculating ν from (41), and then letting that calculated value be the ν for the next time step. This is in fact the method used in the 012 CA code, with several slight modifications; but it has the drawback of involving a larger-than-desired number of floating point operations, which offsets some of the speed advantage of the Boolean update rule. The modifications in the actual code are all aimed at making the number of such operations as small as possible; but it is worthwhile to ask whether there might

not be some other way of making ν assume the correct value which is even faster, perhaps Boolean in character. Such a 'mask automaton' has not, as of this writing, been made to work, but prospects for constructing a viable one seem good as of this writing.

The actual 012 CA code does use equation (41) to calculate ν, after calculating R and n_2 for equilibration using (39) and (40). It then uses the new value of ν to 'seed' the process mask for the inverse $p = 1$ process before the next update step. However, the code only re-calculates ν if the fluid variables ρ and ϵ have changed since the last time step; if they are the same, then the masks are not re-seeded in order to save operations. Unavoidable fluctuations in the random number generator mean that the actual value of ν obtained will not be exactly equal to the calculated value for equilibration; however, the difference is typically less that 1 percent for the lattice size worked with here, and will become smaller as the number of lattice sites gets larger. Another reason for not re-seeding the masks every time step is that we wish to measure the equlibration time of the system after an initial perturbation; thus, after ρ and ϵ have been changed and a new mask created, the mask is left constant for a number of steps in order to let the system relax. The hope is that this relaxation will take place in a number of time steps which is fairly small compared to the number of time steps required for a particle to cross the 'lattice cell'; this latter time is taken to approximate the hydrodynamic time scale (since the lattice cell is considered to be a point in the hydrodynamic system), and we need to have the CA relax to $g = 1$ on a time scale which is short compared to the hydrodynamic time scale in order to justify the change to continous variables which was necessary to derive the hydrodynamic equations in section 2.2.

5 Results of Code Simulations

The detailed structure of the computer code will not be discussed here, since understanding it is not essential to interpreting the results of the simulation. A fairly detailed discussion of the code can be found in Appendix 1. The basic plan, once the code for the update rule had been tested and found to conserve mass, momentum, and energy in a fully closed system (the most basic possible test), was to attach the necessary input and output machinery so that the value of g and any other parameters of interest could be monitored, and then to perturb the system by changing ρ and ϵ and observe relaxation to thermal equilibrium and $g = 1$. An output of the 'effective' ν (as calculated from the actual process masks) versus the calculated ν for equilibration was also generated, but it conveys little information beyond the fact that ν was within 1 percent of its calculated value (quite acceptable considering the unavoidable fluctuations in the random number generator).

Output from several runs of the code is shown in Figures 4 through 7. For each run there is a series of three graphs: first, a plot of the actual perturbations created in the system in terms of the mass density ρ and energy density ϵ present on the lattice; second, a plot of g versus time, in which the perturbations and relaxations can be observed; and third, a plot of the two temperatures T_{12} and T versus time, so that thermal equilibration can be verified. The lattice size used for all simulations was 64 by 66 sites, which is the typical desired size for a 'lattice cell' in a large-scale hydrodynamic simulation; thus, the hydrodynamic space scale is taken to be 64 lattice sites long, and the time scale to be 64 CA time steps long. The fact that the equilibration times observed were 10 percent or less of the hydrodynamic

scale supports the assumptions underlying the change from discrete to continuous variables necessary to developing the hydrodynamic equations.

5.1 Run 1 (Figure 4)

This run is the longest in time of the runs given here, and exhibits most of the essential features of the automaton, being a representative example of several similar runs of the code. The density and energy perturbations, as can be seen from Figure 4, are sinusoidal. The 'noise' at the beginning and end of the plot of g is probably due to statistical effects (see section 5.3 below), which are dependent on the density; as ρ increases towards the middle of the run, the noise is observed to damp out and the value of g remains much closer to 1. It should be noted that the effective ν on the lattice, determined by the process masks, will not be exactly equal to the calculated ν required for equilibration; the error is due to statistical fluctuations in the random number generator and is typically 1 percent for the lattice size of these simulations. The error, of course, decreases as the number of lattice sites increases. This is one of several statistical effects on the system behavior; another such effect is investigated in section 5.3 below.

Also, from both the g and temperature plots the equilibration time of the system is observed to be well under 10 time steps, which is the interval at which perturbations were applied (the sharp 'spikes' in the g and temperature plots show the occurrence of the perturbations, as does the plot of ρ and ϵ). The two temperatures T_{12} and T are not exactly the same throughout the run, but except for the 'spikes' they are close enough to make any variations probably statistical in nature, so that the width of the spikes themselves can be taken as giving the equilibration time. Since the spike width is considerably less than the spacing between the spikes, we

know that the equilibration is taking place in well under 10 time steps; a tentative average would be 4 or 5 time steps, which is under 10 percent of the 64-time step hydrodynamic scale. A similar equilibration time is observed in run 2 (Figure 5).

5.2 Run 2 (Figure 5)

This run is interesting for two reasons, the first of which is that the densities achieved were well above those at which the low-density limit used to derive the $g = 1$ equation in section 4.2 is valid. The units of ρ and ϵ on the plots shown are particles per site and energy units (energy of type 1 particle = 1) per state (recall equations (23) and (24) and their different normalizations); thus the maximum value for ρ is 9.0, and for ϵ, 3.0. Values of these parameters in the first run reached more than 80 percent of maximum, so that the low-density theoretical calculations of sections 4.2 and 4.3 should not have applied. However, the plot of g versus time and temperature versus time show that thermal equilibration is nearly perfect and g is still 1 within 2 percent after equilibration, so that even in regimes outside that of strict theoretical validity the code simulation does a good job of achieving its goal. Study of the g plot will reveal that the equilibrated value of g seems to be decreasing with the increasing densities; this can be understood by taking higher-order corrections into account in our calculations, but only at the cost of much more work, and it is refreshing to find that we can make a much simpler calculation and still obtain results which are quite accurate.

The second interesting aspect of this run is the achievement of a negative-temperature state after time step 10. This occurs, as is discussed in reference [?], because the system has an upper bound for its total energy, corresponding to a state in which n_1 and n_2 are both 1 (all type 1 and 2 states filled). Such a state

has no real meaning in a hydrodynamic simulation; however, its occurrence here opens up the possibility of modeling highly degenerate systems using a CA similar to the one described here. In terms of the equilibrium equations of section 4.3, such a state satisfies the $g = 1$ condition by having an 'inverted population' in which $n_2 > n_1 > n_0$, rather than the reverse, which would be the normal case for a hydrodynamic problem. This corresponds to the ratio $R = n_0/n_1 = n_1/n_2$ being less than 1, and the condition for this, derived from equation (39), is

$$\frac{\rho}{\epsilon} < 3. \tag{45}$$

This condition can readily be verified from examining the plot of ρ and ϵ in Figure 5.2. Such a state is stable as long as the system is isolated and cannot exchange energy with the outside world, which is just another way of saying that there are only a finite number of states that can be occupied, and hence an energy upper bound.

5.3 Runs 3 and 4 (Figures 6 and 7)

These runs were done mainly to investigate the 'noise effect' of statistical fluctuations on the system behavior. The perturbations on ρ and ϵ were very small for both runs, so that the system could be treated as virtually closed while still allowing the routine which re-calculates ν and re-seeds the masks to work (since it only changes the masks if ρ and ϵ change). As is obvious from examining Figures 6 and 7, the fluctuations affect g to a considerable extent (as much as 12 percent fluctuation either way from $g = 1$), but affect the temperature to a lesser extent. Also, by comparing the figures, we find that as the density and energy increase, the amplitude of the g fluctuations grows smaller.

We can justify this on the basis of a simple argument; recall the formula for g

$$g = \frac{\rho n_2}{\epsilon^2} \tag{46}$$

and let g be perturbed by a small amount δg from its equilibrium value of 1, thus:

$$g = 1 + \delta g \tag{47}$$

Then substitute (47) into (46) and arrive at an expression for δg:

$$\delta g = \frac{\rho}{\epsilon^2} n_2 - 1. \tag{48}$$

If we let n_2 also be perturbed by a small amount δn_2 from its equilibrium value of ϵ^2/ρ, we find that

$$\delta g = \frac{\rho}{\epsilon^2} \delta n_2. \tag{49}$$

If we now take the 'standard deviation' of g by the usual methods of statistics, we find that it is given by (49), with the standard deviation of n_2 replacing δn_2. We assume that the standard deviation of n_2 will be fairly constant with ρ and ϵ, and therefore the amplitude of fluctuations in g will vary as ρ/ϵ^2. The values of this are: for run 3, $\rho/\epsilon^2 \approx 117$; for run 4, $\rho/\epsilon^2 \approx 20$. The observed standard deviations can be approximated by $\frac{1}{\sqrt{2}}(g_{max} - g_{min})$ for this rough calculation, which yields: for run 3, $\sigma \approx 0.14$; for run 4, $\sigma \approx 0.042$.

For the formula (49) to be valid, we should have the standard deviations in the same ratio as the ρ/ϵ^2 factors; the actual ratios are 5.8 for the factors and 3.3 for the deviations, which suggests that the effect of changes in n_2 is mitigating the ρ/ϵ^2 effect somewhat. However, the main observation, that increasing ρ and ϵ decreases the fluctuation amplitude, is supported by our calculation here. Furthermore, we

can say that it is necessary to increase both ρ and ϵ in order to achieve this effect; simply increasing ρ by itself (for example by adding more stopped particles to the system) will actually *increase* the fluctuation amplitude, according to (49). Thus, the fluctuations decrease in amplitude as more particles of all types are added to the system in equal ratios. We can observe this effect in run 1 as well: the 'noise' at the beginning and end of the run, as plotted in Figure 4, has $\rho/\epsilon^2 \approx 60$ and $\sigma \approx 0.13$, so that it fits very well with the values from run 4, although not so well with those of run 3. In the middle of run 1, the value of ρ/ϵ^2 is down to 4, and the fluctuations are hardly observable.

5.4 Conclusions

The Galilean invariance code developed in this thesis does ensure $g = 1$ to a degree of accuracy commensurate with the inherent statistical errors in the system. Most of these errors will become negligible as larger and larger systems are dealt with; the assumptions underlying the theoretical development were meant to apply to the hydrodynamic system as a whole and not to a single 'lattice cell' which is meant to simulate a point or infinitesimal element of the fluid. The fact that already the deviations in g (as calculated in section 5.3) are not much higher than 10 percent, when only a single lattice cell is being considered, indicates that over many lattice cells the $g = 1$ condition would hold quite well on a macroscopic scale.

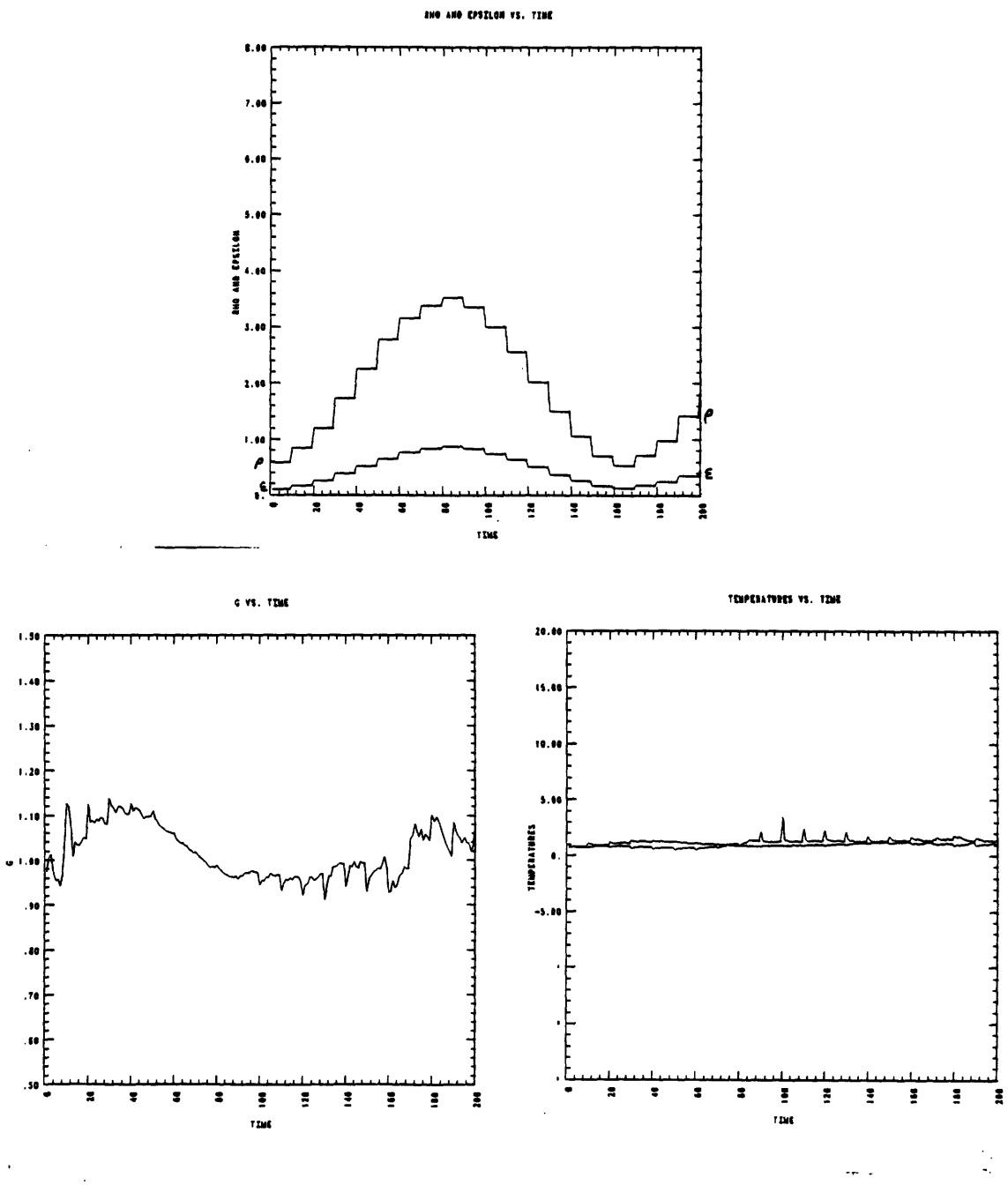

Figure 4: Code Run 1 Output

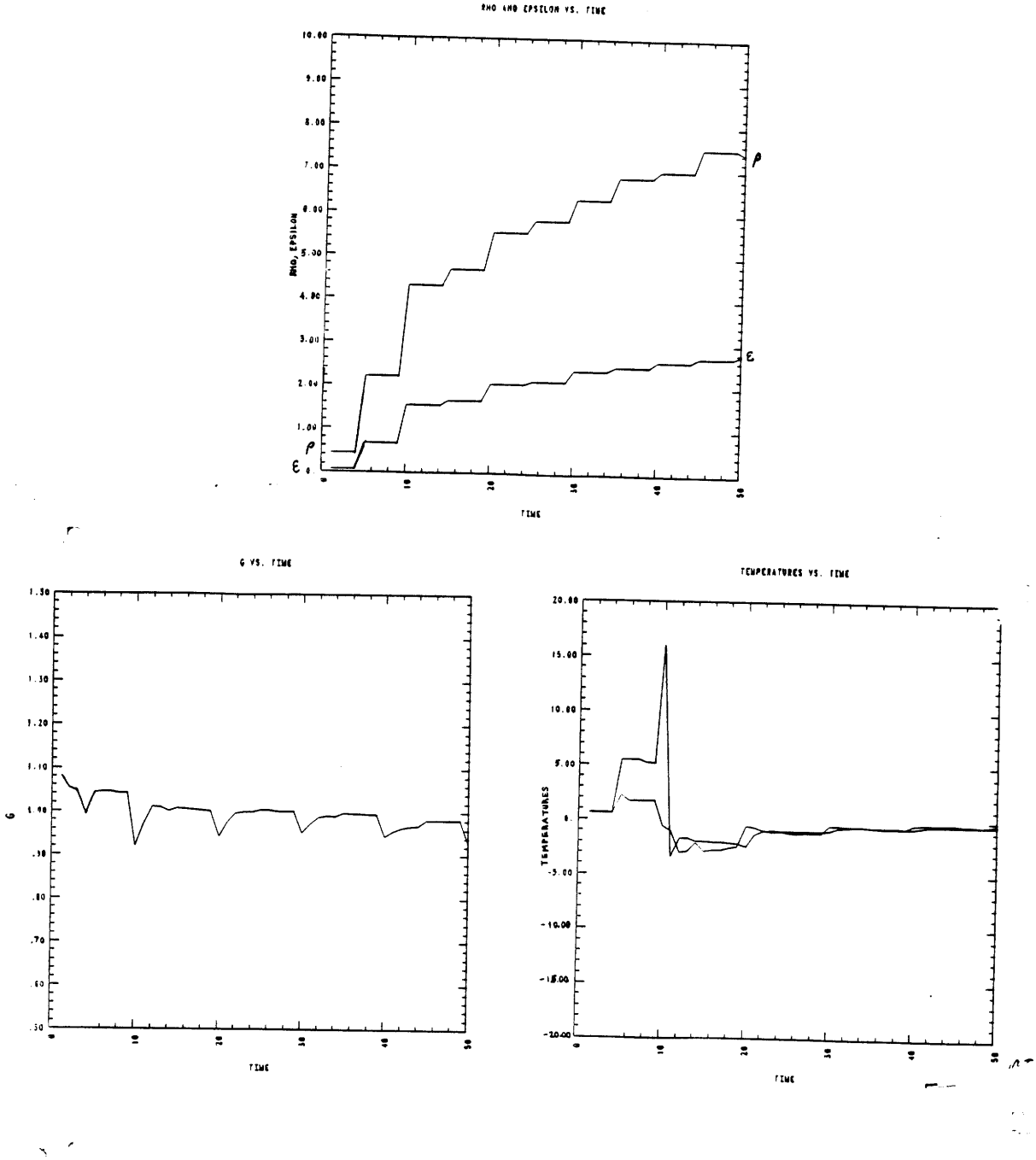

Figure 5: Code Run 2 Output

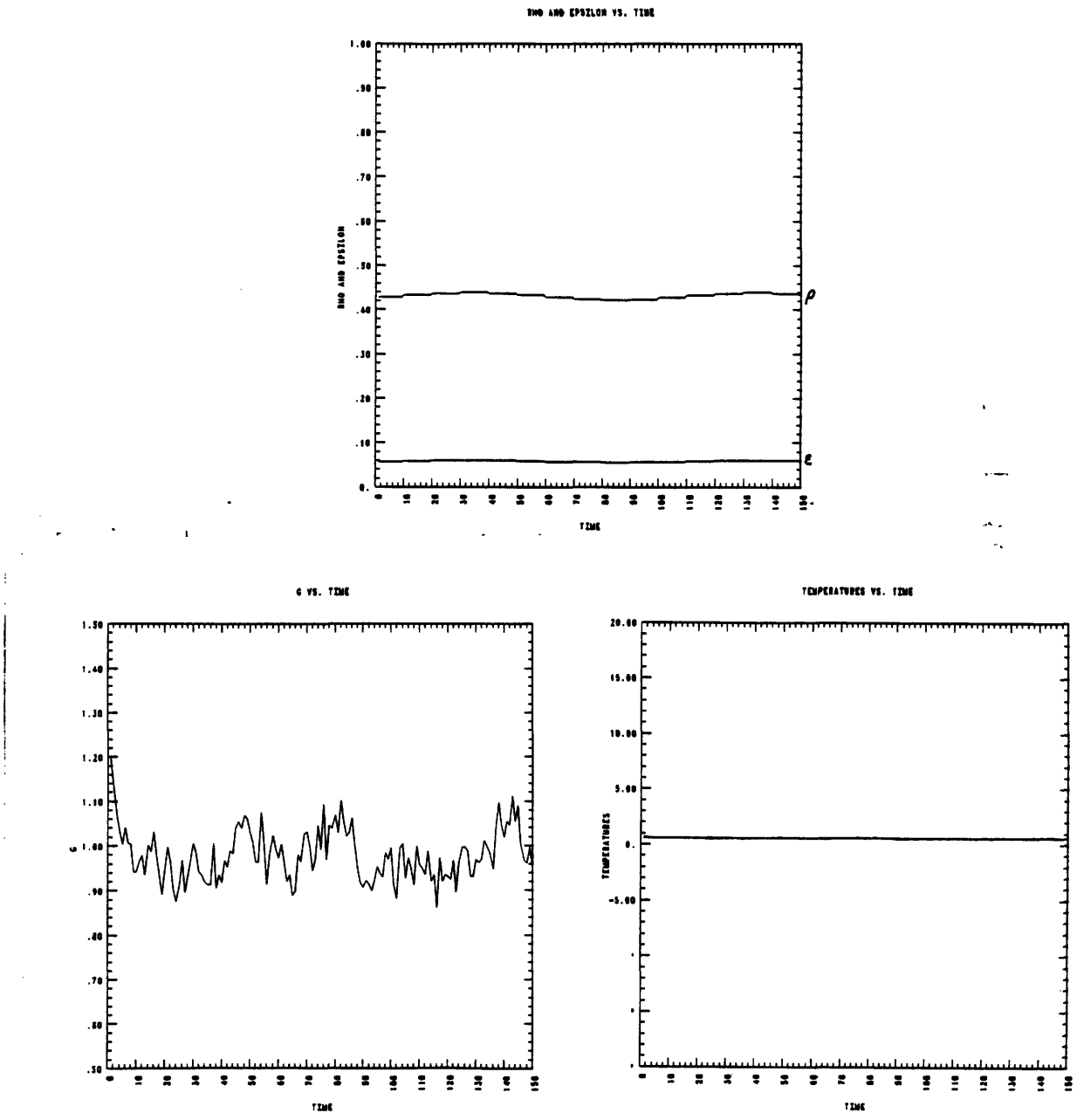

Figure 6: Code Run 3 Output

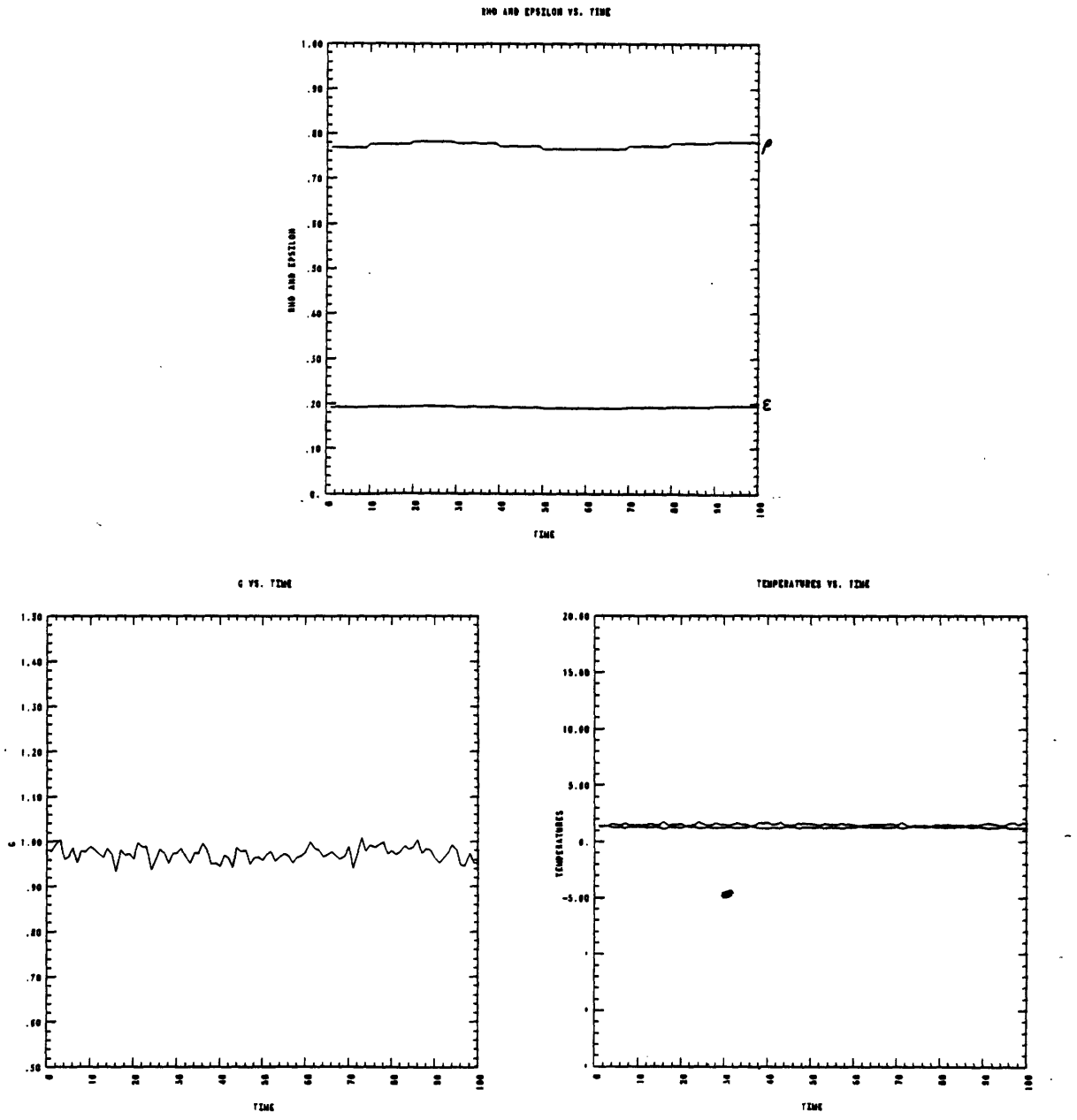

Figure 7: Code Run 4 Output

APPENDICES

A Remarks on the Code

A listing of the FORTRAN code used to realize the CA developed in this thesis is provided in Appendix C. Here a few remarks on its logical structure and adaptations made in writing it will be made. The code is written in seven separate 'modules', each containing one or more subroutines which are intended to accomplish a given task. The first module, 'program newca', is the main program which calls all the subroutines and orchestrates the running of the code. Its structure is very simple, which is why the module method has been used in the code: input is read in, the array is initialized, and the 'output' routine called to calculate system parameters. Then the 'time step loop' is entered; at each step the move routine, the mask initialization routine, the update rule, and the output routine are called in succession, so that the lattice is run through one update cycle and the new system parameters calculated for use in the next cycle. Finally, after the last time step, the output routine is called again to finish things up. The conditional statement in the time step loop is to determine whether particles will be added or subtracted from the system in the move routine.

The Boolean variables $n_{ji}(x,t)$ are stored in 'words' of 32 bits each in the computer memory; the exact 'arrangement' of the words on the lattice is not terribly important, since it affects only the move subroutine (q.v.). The 32-bit words mean that the update rule actually 'processes' 32 lattice sites at a time; this parallelism is another advantage of using Boolean variables for simulations. The words are 'aligned' horizontally except for the left and right walls, which are aligned vertically; therefore, the lattice must be a multiple of 32 sites long in the vertical direction and a multiple of 32 plus 2 extra sites in the horizontal direction. The vertical wall words are mainly there to make the move loop easier.

The six 'modules' are each briefly described below:

Input Module: Contains the subroutine 'input', which reads in program parameters from a data file and from the keyboard, opens all necessary output files, and

calculates parameters necessary for other subroutines. Most of the parameters have to do with proper referencing of array variables.

Initialization Module: Contains the subroutines 'init' and 'mask'; the first initializes the particle variables $n_{ji}(x,t)$, the sign function S_j^p, and the index function $\mathcal{F}^p_{j,i,i'}$; it also sets all process masks except for the inverse $p = 1$ mask to 1 everywhere. The second subroutine initializes the inverse $p = 1$ mask if given a value of ν in the variable 'xnew' which is different from the value 'xnu0' from the previous time step.

Move Module: This subroutine, 'move', is the most complex of the routines in the code; its purpose is to 'propagate' the particles according to their velocity vectors $\vec{c}_{ji}$. In order to do this, system subroutines and functions which can 'circular shift' bits in a word of memory, set bits to 1 or 0, and move bits from one word to another, are utilized. At the end of the move routine is a section to add or remove particles, if that is called for; the condition is that the variable 'iset' be nonzero for particles to be added or subtracted. The number of particles (bits) added or subtracted from each word is given by the variable 'jtest'.

Update Module: This contains the subroutine 'update', which actually realizes the CA update rule by translating the Boolean equations directly into FORTRAN statements, using special system functions which perform logical operations directly on bits of storage. The array reference variables have been chosen to match as closely as possible the indices in the formalism; thus i is used for direction, j for particle type, and p for process index. The functions S_j^p and $\mathcal{F}^p_{j,i,i'}$ (stored in the variables 's' and 'findex') are presumed given in the subroutine (they are initialized by the 'init' routine).

Output Module: Contains the routines 'output' and 'maskcount'; the latter simply counts the number of bits which are 1 in all the inverse $p = 1$ process masks on the lattice and returns that value so that the output routine can calculate the 'effective' ν on the lattice. The output routine can take several paths depending on whether it is called with a positive, zero, or negative argument (corresponding to the middle, end, and beginning of the code); each path outputs only those variables necessary for the section of the code in which it is called. If called in the time step loop it will send to plot files data for the plots of g, temperature, ρ, and ϵ versus time. It also, unless called at the end of the code, calculates the required ν for

equilibration based on the present values of ρ and ϵ, and returns that value in the variable 'xnew' for use by the mask initialization routine.

Counting Module: This contains three subroutines: 'count', 'wallcount', and 'countbits'. The first is the 'main' counting subroutine, which calls the other two; its job is, given the total lattice state, to count the number of particles of each type on the lattice and return them in the array 'inumber', indexed by j (from 0 to 2). The wallcount routine simply counts the particles at the walls (or edge words) of the lattice, since those words are referenced somewhat differently from the interior words. The countbits routine, given an input word of memory, will return in the variable 'icount' the number of bits in that word which are 1.

B The Index Function

The simple forms for the creation and annihilation sums C_{ji} and A_{ji} in the generalized CA update rule developed in section 3 is made possible in large part by the introduction of the 'index function' $\mathcal{F}^{p}_{j,i,i'}$ as a method of selecting the proper permutations of particle species indices j, i and truth function indices p, i'. The purpose of this appendix is to explicitly derive the form of $\mathcal{F}^{p}_{j,i,i'}$ for the 012 CA, as an example of how the process might be done in general. The straightforward method of determining $\mathcal{F}^{p}_{j,i,i'}$ is simply to systematically examine all permutations of its indices and pick out those which should return a value of 1. Obviously this method will become cumbersome for large numbers of processes and particle directions such as will be present in a full hydrodynamic CA; however, we will find as we go that there are definite symmetries in the truth functions which make the task considerably easier.

We begin, then, with the $p = 1$ process, which since it involves multiple particle types is expected to have the most permitted permutations. For this process we first observe that the $j = 0$ particle is defined to have 'direction' $i = 0$ for convenience in the formalism; therefore the permutations $j = 0, i \neq 0$ are at once disallowed. For $i = 0$, however, all of the four truth function indices i' must be included, since all of them involve the stopped particle (it makes no difference in which direction the type 1 and 2 particles move, they will produce the same stopped particle since there is only one at a given site). Therefore, the permutations which give $\mathcal{F}^{p}_{j,i,i'} = 1$ for $p = 1, j = 0$ are: $i = 0$, all i'. For the $j = 2$ particle, we have an 'opposite' situation: all i indices may be permitted, but for each i the truth function index i' must be the same as the particle direction i, so that the permutations which give $\mathcal{F}^{p}_{j,i,i'} = 1$ for $p = 1$, $j = 2$ are: all i, and $i' = i$.

The $j = 1$ particle introduces a new subtlety. Since two particles of this type are involved in the $p = 1$ interaction, we should have two truth function indices i' for each particle direction i. The question then becomes: which two? Obviously one of them is $i' = i$ from the truth function

$$T^{1}_{i'} = n_{1,i'}\, n_{1,i'+1}\, \overline{n}_{2,i'}\, \overline{n}_{0,0}; \tag{50}$$

however, we must be careful not to be misled by (50) into choosing $i' = i + 1$ for the second permitted permutation. I have deliberately written i' in (50), instead of i as was done in developing the formalism of section 3, in order to make it clear that the truth function index is meant, not the particle index. We must then choose the permutation so that both 'functions' of i' in the indices for the type 1 particles will yield i. The choice $i' = i$ is one, but the other is $i' + 1 = i$, which is then reversed to

$i' = i - 1$, or, since we are adding and subtracting modulo 4, $i' = i + 3$. The allowed permutations for $p = 1, j = 1$ are therefore: all i, and $i' = i$ or $i' = i + 3$.

The 'self-collision' processes (so called because they only involve one particle type–they do, however, involve multiple particle 'species', as defined in section 2.2) are much simpler to deal with, since we can immediately eliminate two out of the three possible j indices for each p. The two cases are exactly analogous, so we shall consider the $p = 2$, $j = 1$ case and then generalize to both cases. For this case, we might be tempted at first to say that there should be two truth function indices for each particle direction which will give $\mathcal{F}^{p}_{j,i,i'} = 1$, by a similar argument to that just used for the $p = 1$, $j = 1$ case; the two permutations would then, by an analogous procedure, be found to be $i' = i$ and $i' = i + 2$. However, if we examine the truth function

$$T^2_{i'} = n_{1,i'}\, n_{1,i'+2}\, \overline{n}_{1,i'+1}\, \overline{n}_{1,i'+3}, \tag{51}$$

we find that it possesses the symmetry properties

$$T^2_{i'} = T^2_{i'+2} = \overset{*2}{T}_{i'+1} = \overset{*2}{T}_{i'+3}, \tag{52}$$

so that the two functions given by the permutation we found are actually one and the same. We can therefore not include both in $\mathcal{F}^{p}_{j,i,i'}$, because of the necessity of avoiding 'double counting' of processes; there is really only one process of which the variable n_{ji} is an 'argument' (in the sense defined in section 3.4), but because of symmetries it has several different truth function expressions. Therefore, we can only select one permutation of i, i' to give $\mathcal{F}^{p}_{j,i,i'} = 1$; we choose $i' = i$ for simplicity, but it should be noted that we could just as well choose $i' = i + 2$ without affecting the consistency of the update rule in any way.

The final summary of $\mathcal{F}^{p}_{j,i,i'}$ for the self-collision processes then proceeds as follows: for $p = 2, 3$, $\mathcal{F}^{p}_{j,i,i'} = 1$ if: $j = p - 1$ (made possible by the convenient choice of indices; this is not a fundamental property), and $i' = i$ (for all i). The complete structure of $\mathcal{F}^{p}_{j,i,i'}$ is therefore as follows:

$$\mathcal{F}^{p}_{j,i,i'} = 1 \text{ if and only if:}$$

$$\begin{aligned} p = 1 \text{and:} \quad & j = 0, i = 0 (\text{all } i'), \\ & j = 1, i' = i \text{or} i' = i + 3, \\ & j = 2, i' = i; \\ p = 2, 3 \text{and:} \quad & j = p - 1, i' = i. \end{aligned}$$

C FORTRAN Code List

Main Program

```
      program newca
      real*8 pi
      pi=3.1415926535898
      write(*,1111)
      read(*,*)itmax,itest,jtest0,xfreq
      xj=real(jtest0)
      xf=pi*xfreq
      call input
      call init
      call output(-1)
      write(6,2223)
      isign=1
      do 50 istep=1,itmax
      iset=0
      if(mod(istep,itest).eq.0)then
         iset=1
         jtest=nint(xj*sin(xf*real(isign)))
         write(*,*)jtest
         isign=isign+1
      endif
      call move(iset,jtest)
      call mask
      call update
      call output(istep)
   50 continue
  999 write(6,2224)
      call output(0)
      stop 'THANKS FOR THE MEMORIES!'
 1111 format(/' INPUT MAX. TIME STEPS, ITEST, JTEST0, XFREQ')
 2223 format(///' START EXECUTION.')
 2224 format(///' STOP EXECUTION.')
      end
```

Common Blocks: These are all referenced via 'include' statements in the program modules.

```
      common /calc/ inumber(0:2)
      common /cint/ n(1:2,0:3,1:32768),nstop(32768),sitemax
      common /cwall1/ l1(32768),l3(32768),l5(32768),l7(32768)
      common /cwall2/ l2(32768),l4(32768),l6(32768),l8(32768)
      common /cwalll/ lleft2(32),lleft3(32),lleft4(32)
      common /cwallr/ lright6(32),lright7(32),lright8(32)
      common /parameters/ jm,im,in,im1,im2
      common /startup/ xd0,xd1,xd2,xnu0,seed
      common /storage/ sitenumber,swallnumber
      common /updt/ pmask(3,32768),pmaski(3,32768),s(1:3,0:2)
```

Input Module

```
      subroutine input
      implicit integer*4 (l-t)
      common /master/ nclosure
      include 'cint.for'
      include 'storage.for'
      include 'parameters.for'
      include 'startup.for'
      open(unit=9,file='thesis.dat',status='unknown')
      open(unit=6,file='thesis.out',status='new')
      write(*,8990)
      read(*,*)seed
      read(9,*)imax,jmax
      read(9,*)nplot
      if(nplot.eq.0)goto 8499
      open(unit=10,file='g.dat',status='new')
      open(unit=11,file='nu.dat',status='new')
      open(unit=12,file='temp.dat',status='new')
      open(unit=13,file='dens.dat',status='new')
 8499 read(9,*)xd0,xd1,xd2
 8500 jm=(jmax-2)/32
```

```
      im=imax*jm
      in=imax/32
      im1=im-jm
      im2=1+jm
      sitemax=im1-jm
      sj1=sitemax-jm
      sj2=im2
      sitenumber=sitemax*32
      swallnumber=(2*(imax+jmax))-4
      write(6,5500)
      write(6,5501)imax,jmax
      write(6,5551)jm,im,in
      write(6,5553)xd0,xd1,xd2
c
      dinum=real(sitenumber)
      dfactor=real(imax*jmax)/dinum
      dwfactor=real(swallnumber-4)/dinum
      xd0=xd0*dfactor
      xd1=xd1*(1.0+(dwfactor/4.0))
      xd2=xd2*(1.0+(dwfactor/2.0)+(1.0/dinum))
c
      return
 5500 format(/' OUTPUT FILE FOR CA PROGRAM')
 5501 format(///i5,' by ',i5,' total array sites.')
 5551 format(/i5,' words per row,',i5,' words, and',i5,' side words.')
 5553 format(/'  Input occupation numbers: ',4f6.3)
 8990 format(/' INPUT RANDOM NUMBER SEED')
      end
```

Initialization Module

```
      subroutine init
      implicit integer*4 (1-t)
      include 'updt.for'
      include 'cint.for'
      include 'startup.for'
      do 10 k=1,sitemax
```

```
      do 10 kk=0,31
      ycheck=ran(seed)
      if(ycheck.gt.xd0)goto 13
      nstop(k)=ibset(nstop(k),kk)
   13    do 15 i=0,3
         ycheck=ran(seed)
         if(ycheck.gt.xd1)goto 11
         n(1,i,k)=ibset(n(1,i,k),kk)
   11    ycheck=ran(seed)
         if(ycheck.gt.xd2)goto 15
         n(2,i,k)=ibset(n(2,i,k),kk)
   15    continue
   10 continue
c
      do 25 p=1,3
      do 25 j=0,2
   25 s(p,j)=0
      s(1,2)=not(s(1,2))
      s(1,0)=not(s(1,0))
c
      do 26 p=1,3
      do 26 j=0,2
      do 26 i=0,3
      il=mod(i+3,4)
      do 26 ii=0,3
      findex(p,j,i,ii)=0
      if(p.eq.1)then
        if((j.ne.0).and.(i.eq.ii))findex(p,j,i,ii)=
     +                  not(findex(p,j,i,ii))
        if((j.eq.1).and.(ii.eq.il))findex(p,j,i,ii)=
     +                  not(findex(p,j,i,ii))
      endif
      if(p.eq.2)then
        if((j.eq.1).and.(i.eq.ii))findex(p,j,i,ii)=
     +                  not(findex(p,j,i,ii))
      endif
      if(p.eq.3)then
        if((j.eq.2).and.(i.eq.ii))findex(p,j,i,ii)=
```

```
     +                    not(findex(p,j,i,ii))
        endif
   26   continue
        do 27 ii=0,3
        findex(1,0,0,ii)=not(findex(1,0,0,ii))
   27   continue
c
        psset=0
        psset=not(psset)
        do 29 k=1,sitemax
        pmask(1,k)=psset
        pmask(3,k)=psset
        pmaski(3,k)=psset
        pmask(2,k)=psset
        pmaski(2,k)=psset
   29   continue
        xnu0=0.01
        return
        end
c
        subroutine mask
        implicit integer*4 (1-t)
        include 'updt.for'
        include 'cint.for'
        include 'startup.for'
        include 'storage.for'
        if(xnew.eq.xnu0)goto 39
        xnu0=xnew
        do 30 k=1,sitemax
        pdset=0
           do 35 ibit=0,31
           ycheck=ran(seed)
           if(ycheck.gt.xnu0)goto 35
           pdset=ibset(pdset,ibit)
   35      continue
        pmaski(1,k)=pdset
   30   continue
   39   return
```

```
      end
```

Move Module

```
      subroutine move(iset,jtest)
      implicit integer*4 (l-t)
      logical*4 bitvalue
      dimension ln1(32768),ln2(32768),ln3(32768),ln4(32768)
      dimension ln5(32768),ln6(32768),ln7(32768),ln8(32768)
      dimension lleft6(32),lleft7(32),lleft8(32)
      include 'cint.for'
      include 'parameters.for'
      include 'cwall.for'
      do 800 i=im2,im1
      k=i-jm
      l1(i)=n(1,0,k)
      l2(i)=n(2,0,k)
      l3(i)=n(1,1,k)
      l4(i)=n(2,1,k)
      l5(i)=n(1,2,k)
      l6(i)=n(2,2,k)
      l7(i)=n(1,3,k)
      l8(i)=n(2,3,k)
 800  continue
      do 700 i=1,im1
      shift=i+jm
      ln6(i)=ishftc(l6(shift),-1,32)
      ln5(i)=l5(shift)
      ln4(i)=ishftc(l4(shift),1,32)
 700  continue
      do 710 i=im2,im1
      ln3(i)=ishftc(l3(i),1,32)
      ln7(i)=ishftc(l7(i),-1,32)
 710  continue
      do 720 i=im2,im
      shift=i-jm
      ln2(i)=ishftc(l2(shift),1,32)
```

```
      ln1(i)=l1(shift)
      ln8(i)=ishftc(l8(shift),-1,32)
720   continue
      do 725 i=1,in
      lleft2(i)=ishftc(lleft2(i),1,32)
      lleft4(i)=ishftc(lleft4(i),-1,32)
      lright6(i)=ishftc(lright6(i),-1,32)
      lright8(i)=ishftc(lright8(i),1,32)
725   continue
      do 726 i=1,in-1
      shift=i+1
      call mvbits(lleft4(shift),31,1,lleft4(i),31)
      lleft4(shift)=ibclr(lleft4(shift),31)
      call mvbits(lright6(shift),31,1,lright6(i),31)
      lright6(shift)=ibclr(lright6(shift),31)
      ip=in-i
      ip1=ip+1
      call mvbits(lleft2(ip),0,1,lleft2(ip1),0)
      lleft2(ip)=ibclr(lleft2(ip),0)
      call mvbits(lright8(ip),0,1,lright8(ip1),0)
      lright8(ip)=ibclr(lright8(ip),0)
726   continue
      do 730 i=1,in
      kk=(i-1)*32
      do 730 j=1,32
      k=kk+j
      ikjm=k*jm
      ik1=ikjm-jm+1
      jd=j-1
      if(k.eq.imax)goto 731
      call mvbits(ln4(ikjm),0,1,lright4(i),jd)
      call mvbits(ln6(ik1),31,1,lleft6(i),jd)
      if(k.eq.1)goto 730
      call mvbits(ln3(ikjm),0,1,lright3(i),jd)
      call mvbits(ln7(ik1),31,1,lleft7(i),jd)
731   call mvbits(ln2(ikjm),0,1,lright2(i),jd)
      call mvbits(ln8(ik1),31,1,lleft8(i),jd)
730   continue
```

```
      do 740 i=im2,im1
      i1=im-i
      i11=i1+1
      id=i-1
      if(mod(i,jm).eq.0)goto 745
      call mvbits(ln4(i1),0,1,ln4(i11),0)
      call mvbits(ln3(i1),0,1,ln3(i11),0)
      call mvbits(ln2(i1),0,1,ln2(i11),0)
      if(mod(i,jm).eq.1)goto 740
745   call mvbits(ln8(i),31,1,ln8(id),31)
      call mvbits(ln7(i),31,1,ln7(id),31)
      call mvbits(ln6(i),31,1,ln6(id),31)
740   continue
      do 750 i=1,jm-1
      itop=im1+i
      jtop=im-i
      jbot=jm-i
      i1=i+1
      itop1=itop+1
      jtop1=jtop+1
      jbot1=jbot+1
      call mvbits(ln4(jbot),0,1,ln4(jbot1),0)
      call mvbits(ln2(jtop),0,1,ln2(jtop1),0)
      call mvbits(ln6(i1),31,1,ln6(i),31)
      call mvbits(ln8(itop1),31,1,ln8(itop),31)
750   continue
      do 760 i=1,in
      kk=(i-1)*32
      do 760 j=1,32
      k=kk+j
      ikjm=k*jm
      ik1=ikjm-jm+1
      jd=j-1
      if(k.eq.1)goto 761
      call mvbits(lleft2(i),jd,1,ln2(ik1),0)
      call mvbits(lright8(i),jd,1,ln8(ikjm),31)
      if(k.eq.imax)goto 760
      call mvbits(lleft3(i),jd,1,ln3(ik1),0)
```

```
      call mvbits(lright7(i),jd,1,ln7(ikjm),31)
 761  call mvbits(lleft4(i),jd,1,ln4(ik1),0)
      call mvbits(lright6(i),jd,1,ln6(ikjm),31)
 760  continue
c
      do 300 i=1,in
      lleft2(i)=lleft6(i)
      lleft6(i)=0
      lleft3(i)=lleft7(i)
      lleft7(i)=0
      lleft4(i)=lleft8(i)
      lleft8(i)=0
      lright8(i)=lright4(i)
      lright4(i)=0
      lright7(i)=lright3(i)
      lright3(i)=0
      lright6(i)=lright2(i)
      lright2(i)=0
 300  continue
      do 350 i=1,jm
      itop=im1+i
      ln8(i)=ln4(i)
      ln4(i)=0
      ln1(i)=ln5(i)
      ln5(i)=0
      ln2(i)=ln6(i)
      ln6(i)=0
      ln4(itop)=ln8(itop)
      ln8(itop)=0
      ln5(itop)=ln1(itop)
      ln1(itop)=0
      ln6(itop)=ln2(itop)
      ln2(itop)=0
 350  continue
      do 765 i=1,im
      l1(i)=ln1(i)
      l2(i)=ln2(i)
      l3(i)=ln3(i)
```

```
      l4(i)=ln4(i)
      l5(i)=ln5(i)
      l6(i)=ln6(i)
      l7(i)=ln7(i)
      l8(i)=ln8(i)
  765 continue
      do 770 i=im2,im1
      k=i-jm
      n(1,0,k)=l1(i)
      n(2,0,k)=l2(i)
      n(1,1,k)=l3(i)
      n(2,1,k)=l4(i)
      n(1,2,k)=l5(i)
      n(2,2,k)=l6(i)
      n(1,3,k)=l7(i)
      n(2,3,k)=l8(i)
  770 continue
c
      if(iset.eq.0)goto 799
c
      if(jtest.le.0)goto 395
c
      do 389 k=1,sitemax
      jsub0=0
      jsub1=0
      jsub2=0
c
      do 388 ibit=0,31
      if(jsub0.gt.jtest)goto 381
      bitvalue=btest(nstop(k),ibit)
      if(.not.bitvalue)then
        nstop(k)=ibset(nstop(k),ibit)
        jsub0=jsub0+1
      endif
  381   do 386 i=0,3
        if(jsub1.gt.jtest)goto 382
        bitvalue=btest(n(1,i,k),ibit)
        if(.not.bitvalue)then
```

```
            n(1,i,k)=ibset(n(1,i,k),ibit)
            jsub1=jsub1+1
         endif
 382     if(jsub2.gt.jtest)goto 386
         bitvalue=btest(n(2,i,k),ibit)
         if(.not.bitvalue)then
            n(2,i,k)=ibset(n(2,i,k),ibit)
            jsub2=jsub2+1
         endif
 386     continue
 388   continue
 389   continue
c
 395   if(jtest.ge.0)goto 799
       jtest=abs(jtest)
c
       do 399 k=1,sitemax
       jsub0=0
       jsub1=0
       jsub2=0
c
       do 398 ibit=0,31
       if(jsub0.gt.jtest)goto 391
       bitvalue=btest(nstop(k),ibit)
       if(bitvalue)then
          nstop(k)=ibclr(nstop(k),ibit)
          jsub0=jsub0+1
       endif
 391     do 396 i=0,3
         if(jsub1.gt.jtest)goto 392
         bitvalue=btest(n(1,i,k),ibit)
         if(bitvalue)then
            n(1,i,k)=ibclr(n(1,i,k),ibit)
            jsub1=jsub1+1
         endif
 392     if(jsub2.gt.jtest)goto 396
         bitvalue=btest(n(2,i,k),ibit)
         if(bitvalue)then
```

```
            n(2,i,k)=ibclr(n(2,i,k),ibit)
            jsub2=jsub2+1
         endif
 396     continue
 398  continue
 399  continue
c
 799  return
      end
```

Update Module

```
      subroutine update
      implicit integer*4 (1-t)
      integer*4 find1
      include 'updt.for'
      include 'cint.for'
      dimension t(1:3,0:3),ti(1:3,0:3),matrix(1:3,0:3)
      do 100 k=1,sitemax
c
         do 190 i=0,3
         i1=mod(i+1,4)
         t(1,i)=iand(n(1,i,k),iand(n(1,i1,k),
     +                not(ior(n(2,i,k),nstop(k)))))
         ti(1,i)=iand(n(2,i,k),iand(nstop(k),
     +                not(ior(n(1,i,k),n(1,i1,k)))))
  190    continue
         do 191 p=2,3
         jp=p-1
         t(p,0)=iand(n(jp,0,k),iand(n(jp,2,k),
     +                not(ior(n(jp,1,k),n(jp,3,k)))))
         ti(p,0)=iand(n(jp,1,k),iand(n(jp,3,k),
     +                not(ior(n(jp,0,k),n(jp,2,k)))))
            do 195 i=1,3
            t(p,i)=ti(p,i-1)
            ti(p,i)=t(p,i-1)
  195       continue
```

```
  191     continue
c
          do 150 p=1,3
          do 150 i=0,3
          melement=0
             do 155 pp=1,3
             if(pp.eq.p)goto 155
             if((p.gt.1).and.(pp.gt.1))goto 155
             p1=pmask(pp,k)
             pi1=pmaski(pp,k)
             do 155 ii=0,3
             t1=t(pp,ii)
             ti1=ti(pp,ii)
             melement=ior(melement,ior(iand(p1,t1),
     +                       iand(pi1,ti1)))
  155        continue
          pp1=pmask(p,k)
          ppi1=pmaski(p,k)
             do 157 ii=0,3
             if(ii.eq.i)goto 157
             t1=t(p,ii)
             ti1=ti(p,ii)
             melement=ior(melement,ior(iand(pp1,t1),
     +                       iand(ppi1,ti1)))
  157        continue
          matrix(p,i)=not(melement)
  150     continue
c
          do 103 j=0,2
          do 103 i=0,3
          psum=0
          msum=0
             do 104 p=1,3
             s1=s(p,j)
             sbar1=not(s1)
             p1=pmask(p,k)
             pi1=pmaski(p,k)
             do 104 ii=0,3
```

```
            mpi=matrix(p,ii)
            t1i=t(p,ii)
            ti1i=ti(p,ii)
            term1=iand(p1,iand(mpi,t1i))
            term2=iand(pi1,iand(mpi,ti1i))
            find1=findex(p,j,i,ii)
            psum=ior(psum,iand(find1,ior(iand(s1,term1),
     +                    iand(sbar1,term2))))
            msum=ior(msum,iand(find1,ior(iand(sbar1,term1),
     +                    iand(s1,term2))))
  104       continue
          if(j.eq.0)then
            nstop(k)=ior(iand(nstop(k),not(msum)),psum)
            goto 103
          endif
          n(j,i,k)=ior(iand(n(j,i,k),not(msum)),psum)
  103     continue
c
  100  continue
c
  200  return
       end
```

Output Module

```
      subroutine output(icode)
      implicit integer*4 (1-t)
      dimension f(0:2),fhat(0:2)
      include 'storage.for'
      include 'calc.for'
      include 'startup.for'
      common /maskc/ kmasknum
      call count(icode)
      mass=inumber(0)+inumber(1)+inumber(2)
      menergy=inumber(1)+(2*inumber(2))
      xmass=real(mass)
      xenergy=real(menergy)
```

```
c
      sites=sitenumber+swallnumber
      xsnum=real(sitenumber)
      xsites=real(sites)
      density=xmass/xsites
      epsilon=xenergy/(4.0*xsites)
      do 4000 j=0,2
      xstates=real(j*swallnumber+4*sitenumber-4)
      xnumber=real(inumber(j))
      if(j.eq.0)f(j)=xnumber/xsites
      if(j.ne.0)f(j)=xnumber/xstates
      f1=1.-f(j)
      fhat(j)=f(j)/f1
 4000 continue
c
      if(icode.eq.0)goto 1909
c
      gnum=f(2)*(f(0)+(4.0*f(1))+(4.0*f(2)))
      gden=(f(1)+(2.0*f(2)))
      gee=gnum/(gden*gden)
c
      are=(4.0*xmass/xenergy)-2.0
      entwo=(xenergy*xenergy)/(16.0*xmass*xsites)
      if(are*entwo .eq. 1)then
         xnew=0.99998
         goto 1905
      endif
      xnew=1.0-((entwo*(1.0-are)*(1.0-are))/
     +         ((1.0-(are*entwo))**2.0))
c
 1905 if(icode.lt.0)goto 1909
c
      call maskcount
      xnu=real(kmasknum)/xsnum
      if(xnu.le.0)xnu=0.00001
      beta12=log(fhat(1)/fhat(2))
      xtemp12=1./beta12
      beta=log(xnu*(fhat(0)/fhat(1)))
```

```
      xtemp=1./beta
c
      if(icode.gt.0)goto 1910
c
 1909 write(6,1998)
      write(6,1991)mass,menergy
      write(6,4901)sites
      write(6,4902)f(0),f(1),f(2)
      goto 1999
 1910 write(6,1993)icode,gee,xnu0,xnu,xtemp12,xtemp,
     +                 density,epsilon
      if(nplot.eq.0)goto 1999
      write(10,1994)icode,gee,gr2
      write(11,1994)icode,xnu,xnu0
      write(12,1994)icode,xtemp12,xtemp
      write(13,1994)icode,density,epsilon
 1999 return
 1991 format(/' Total mass: ',i7,' -- Total energy: ',i7)
 1992 format(/'  g, T(12,0nu), tmask: ',4f11.8)
 1993 format(' Step: ',i3,1x,f9.5,1x,2f9.5,1x,
     +         2f9.5,1x,2f9.5)
 1994 format(i4,2f20.12)
 1998 format(//,' System properties:')
 4901 format(/' Number of array sites: ',i10)
 4902 format(/' Occupation numbers (0,1,2): ',3f11.8)
      end
c
      subroutine maskcount
      implicit integer*4 (l-t)
      include 'updt.for'
      include 'cint.for'
      common /maskc/ kmasknum,knumber
      common /countup/ icount
      kmasknum=0
      do 5001 k=1,sitemax
      call countbits(pmaski(1,k))
      kmasknum=kmasknum+icount
 5001 continue
```

```
      return
      end
```

Counting Module

```
      subroutine count(icode)
      implicit integer*4 (1-t)
      dimension itotal(1:3,0:3)
      common /cwallc/ iwall(1:2)
      include 'cint.for'
      include 'calc.for'
      common /countup/ icount
      inumber0=0
      do 1050 k=1,sitemax
      call countbits(nstop(k))
      inumber0=inumber0+icount
1050  continue
      inumber(0)=inumber0
      do 1000 j=1,2
      inumber1=0
         do 1010 i=0,3
         itotal1=0
            do 1020 k=1,sitemax
            call countbits(n(j,i,k))
            itotal1=itotal1+icount
1020        continue
         itotal(j,i)=itotal1
         inumber1=inumber1+itotal1
1010     continue
      inumber(j)=inumber1
1000  continue
      call wallcount
      do 1500 j=1,2
      inumber(j)=inumber(j)+iwall(j)
1500  continue
      if(icode.ne.0)goto 1099
      write(6,1599)inumber(0),inumber(1),inumber(2)
```

```
 1099  return
 1599  format(/' Particle counts: ',3i6)
       end
c
       subroutine wallcount
       implicit integer*4(1-t)
       common /cwallc/ iwall(1:2)
       common /countup/icount
c       include parameters.for
c       include cwall.for
       common /parameters/ jm,im,in,im1,im2
       common /cwall1/ l1(32768),l3(32768),l5(32768),l7(32768)
       common /cwall2/ l2(32768),l4(32768),l6(32768),l8(32768)
       common /cwall1/ lleft2(32),lleft3(32),lleft4(32)
       common /cwallr/ lright6(32),lright7(32),lright8(32)
       iwall1=0
       iwall2=0
       do 4000 i=1,jm
       itop=im1+i
       call countbits(l1(i))
       iwall1=iwall1+icount
       call countbits(l8(i))
       iwall2=iwall2+icount
       call countbits(l2(i))
       iwall2=iwall2+icount
       call countbits(l5(itop))
       iwall1=iwall1+icount
       call countbits(l4(itop))
       iwall2=iwall2+icount
       call countbits(l6(itop))
       iwall2=iwall2+icount
 4000  continue
       do 4010 i=1,in
       call countbits(lleft3(i))
       iwall1=iwall1+icount
       call countbits(lleft2(i))
       iwall2=iwall2+icount
       call countbits(lleft4(i))
```

```
      iwall2=iwall2+icount
      call countbits(lright7(i))
      iwall1=iwall1+icount
      call countbits(lright6(i))
      iwall2=iwall2+icount
      call countbits(lright8(i))
      iwall2=iwall2+icount
 4010 continue
      iwall(1)=iwall1
      iwall(2)=iwall2
      return
      end
c
      subroutine countbits(narg)
      implicit integer*4 (l-t)
      common /countup/ icount
      narg1=narg
      icount=0
      nset=1
      do 2000 jjj=1,32
      ntest=iand(narg1,nset)
      if(ntest.eq.0)goto 2010
      icount=icount+1
 2010 narg1=ishft(narg1,-1)
 2000 continue
      return
      end
```